
CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Matemáticas e Informática

**Universidad Politécnica de Madrid**
Escuela Técnica Superior de Ingenieros Informáticos

## TRABAJO FIN DE GRADO

# Manejo de memoria en programación lógica con tabulación

**Autor:** Joaquín Arias Herrero
**Director:** Manuel Carro Liñares

MADRID, JUNIO DE 2014

# RESUMEN

Los lenguajes de programación son el idioma que los programadores usamos para comunicar a los computadores qué queremos que hagan. Desde el lenguaje ensamblador, que traduce una a una las instrucciones que interpreta un computador hasta lenguajes de alto nivel, se ha buscado desarrollar lenguajes más cercanos a la forma de pensar y expresarse de los humanos.

Los lenguajes de **programación lógicos** como Prolog utilizan a su vez el lenguaje de la lógica de $1^{er}$ orden de modo que el programador puede expresar las premisas del problema que se quiere resolver sin preocuparse del cómo se va a resolver dicho problema. La resolución del problema se equipara a encontrar una deducción del objetivo a alcanzar a partir de las premisas y equivale a lo que entendemos por la ejecución de un programa.

**Ciao** es una implementación de Prolog (`http://www.ciao-lang.org`) y utiliza el método de resolución SLD, que realiza el recorrido de los árboles de decisión en profundidad (depth-first) lo que puede derivar en la ejecución de una rama de busqueda infinita (en un bucle infinito) sin llegar a dar respuestas.

Ciao, al ser un sistema modular, permite la utilización de extensiones para implementar estrategias de resolución alternativas como la tabulación (OLDT). La tabulación es un método alternativo que se basa en memorizar las llamadas realizadas y sus respuestas para no repetir llamadas y poder usar las respuestas sin recomputar las llamadas. Algunos programas que con SLD entran en un bucle infinito, gracias a la tabulación dán todas las respuestas y termina.

El modulo *tabling* es una implementación de tabulación mediante el algoritmo CHAT. Esta implementación es una versión beta que no tiene implementado un manejador de memoria.

Entendemos que la gestión de memoria en el módulo de tabling tiene gran importancia, dado que la resolución con tabulación permite reducir el tiempo de computación (al no repetir llamadas), aumentando los requerimientos de memoria (para guardar las llamadas y las respuestas).

Por lo tanto, el objetivo de este trabajo es implementar un mecanismo de gestión

de la memoria en Ciao con el módulo tabling cargado. Para ello se ha realizado la implementación de:

- Un mecanismo de captura de errores que: detecta cuando el computador se queda sin memoria y activa la reinicialización del sitema.
- Un procedimiento que ajusta los punteros del modulo de tabling que apuntan a la WAM tras un proceso de realojo de algunas de las áreas de memoria de la WAM.
- Un gestor de memoria del modulo de tabling que detecta cuando es necesario realizar una ampliación de las áreas de memoria del modulo de tabling, realiza la solicitud de más memoria y realiza el ajuste de los punteros.

Para ayudar al lector no familiarizado con este tema, describimos los datos que Ciao y el módulo de tabling alojan en las áreas de memoria dinámicas que queremos gestionar.

Los casos de pruebas desarrollados para evaluar la implementación del gestor de memoria, ponen de manifiesto que:

- Disponer de un gestor de memoria dinámica permite la ejecución de programas en un mayor número de casos.
- La política de gestión de memoria incide en la velocidad de ejecución de los programas.

# ABSTRACT

Programming languages are the language that programmers use in order to communicate to computers what we want them to do. Starting from the assembly language, which translates one by one the instructions to the computer, and arriving to highly complex languages, programmers have tried to develop programming languages that resemble more closely the way of thinking and communicating of human beings.

**Logical programming languages**, such as Prolog, use the language of logic of the first order so that programmers can express the premise of the problem that they want to solve without having to solve the problem itself. The solution to the problem is equal to finding a deduction of the objective to reach starting from the premises and corresponds to what is usually meant as the execution of a program.

Ciao is an implementation of Prolog (`http://www.ciao-lang.org`) and uses the method of resolution SLD that carries out the path of the decision trees in depth (depth-frist). This can cause the execution of an infinite searching branch (an infinite loop) without getting to an answer.

Since Ciao is a modular system, it allows the use of extensions to implement alternative resolution strategies, such as tabulation (OLDT). Tabulation is an alternative method that is based on the memorization of executions and their answers, in order to avoid the repetition of executions and to be able to use the answers without re-executions. Some programs that get into an infinite loop with SLD are able to give all the answers and to finish thanks to tabulation.

The tabling package is an implementation of tabulation through the algorithm CHAT. This implementation is a beta version which does not present a memory handler.

The management of memory in the tabling package is highly important, since the solution with tabulation allows to reduce the system time (because it does not repeat executions) and increases the memory requirements (in order to save executions and answers).

Therefore, the objective of this work is to implement a memory management mechanism in Ciao with the tabling package loaded. To achieve this goal, the following

implementation were made:

- An error detection system that reveals when the computer is left without memory and activate the reinizialitation of the system.
- A procedure that adjusts the pointers of the tabling package which points to the WAM after a process of realloc of some of the WAM memory stacks.
- A memory manager of the tabling package that detects when it is necessary to expand the memory stacks of the tabling package, requests more memory, and adjusts the pointers.

In order to help the readers who are not familiar with this topic, we described the data which Ciao and the tabling package host in the dynamic memory stacks that we want to manage.

The test cases developed to evaluate the implementation of the memory manager show that:

- A manager for the dynamic memory allows the execution of programs in a larger number of cases.
- Memory management policy influences the program execution speed.

# INDICE

# Capítulo 1

# INTRODUCCION

La evolución de los lenguajes de programación ha buscado el equilibrio entre facilitar la programación de los computadores y no perder rendimiento computacional. Los objetivos principales de estos lenguajes de programación que denominamos de alto nivel, han sido:

- Aproximar el lenguaje al problema que se quiere resolver.
- Implementar mecanismos de gestión de memoria implícita de modo que el programador no tenga que ocuparse de dicha gestión.

Entre estos lenguajes de alto nivel están los lenguajes declarativos. Denominados así porque están basados en una teoría. El lenguaje de programación Prolog (del francés PROgrammation en LOGique), está basado en la lógica de $1^{er}$ orden y es por tanto un lenguaje de programación lógica. Prolog es un lenguaje semi-interpretado que se compila a bytecode y que se ejecuta sobre una máquina virtual, la Warren Abstract Machine (WAM).

Una evolución de Prolog es la programación lógica con tabulación, que incorpora el método de resolución con tabulación. La tabulación consiste en guarda en una tabla las llamadas realizadas (procedimientos ejecutados para buscar una respuesta) y las respuestas obtenidas por dichas llamadas permitiendo:

- Aumentar la eficacia al evitar repetir llamadas de las que ya tenemos las respuestas.
- Evitar entrar en bucles (llamadas recursivas sin fin) pues las llamadas repetidas se suspenden hasta que se tienen las respuestas de la llamada anterior.

Esto a su vez permite aumentar la expresividad del lenguaje.

Ciao Prolog es una implementación de Prolog que gracias a la filosofía modular de Ciao puede extenderse mediante módulos. Uno de estos módulos es **Tabling**, que implementa la resolución con tabulación.

Ciao Prolog como implementación de un lenguaje de alto nivel dispone de un mecanismo de gestión de memoria dinámica que consiste en:

- Crear las áreas de memoria dinámicas en la WAM que se adaptarán a las necesidades del programa que está en ejecución. Las áreas de memoria dinámica de la WAM son el Heap, el Stack, el Choice-Stack y el Trail.
- Cuando alguna de las áreas de memoria se queda sin espacio, se ejecuta el recolector de basura (GC) según algoritmo descrito en [3]. El recolector de basura se encarga de determinar los datos que ya no son útiles, los libera y agrupa el resto de manera que no queden huecos.
  - Si el GC ha generado suficiente espacio se continua con la ejecución del programa.
  - En caso contrario es preciso solicitar un bloque de memoria mayor donde realojar los datos de modo que tengamos espacio libre para continuar con la ejecución del programa.
    * Si los datos se han podido realojar, se hace necesario ajustar los punteros (variable que guarda la direccion de la posición de memoria de los datos), pues la posición en memoria ha cambiado.
    * Cuando el computador no dispone de memoria suficiente para satisfacer la solicitud de un bloque de memoria mayor, el programa se debe detener su ejecución y se emite un mensaje de error.

El módulo Tabling, sin embargo, al tratarse de una versión beta no tiene implementado el mecanismo que ajusta los punteros tras el procedimiento de realojo de alguna de las áreas de memoria de la WAM. Por ello al cargar el modulo Tabling, se amplia el tamaño de las áreas de memoria de la WAM y crean las áreas de memoria para el Tabling de manera estática (cuando se llenan el programa se detiene). Esto hace que programas con escaso requerimiento de memoria desperdicien memoria asignada y que programas con altos requerimientos cuando alguna de las áreas de memoria está llena, no pueda seguir la ejecución aún cuando la computadora tiene memoria disponible.

## 1.1. Objetivo

El objetivo del presente trabajo es hacer compatible el uso del modulo de tabulación con una gestión de memoria dinámica, de modo que la asignación de memoria se

adapte a los requerimientos del programa en ejecución. Para ello se han definido cuatro objetivos:

1. Hacer compatible el módulo de tabulación con la gestión de memoria de la WAM, ajustando los punteros de los datos de las áreas de la tabulación tras la reubicación de las áreas de la WAM.
2. Identificar cuando las áreas de memoria de Tabling están próximas a estar llenas.
3. Aplicar los criterios de gestión de memoria de la WAM a Tabling:
   - Expandiendo las áreas de memoria.
   - Realizando recogida de basura.
4. Abortar la ejecución y reiniciar correctamente Ciao y el módulo Tabling cuando no queda espacio de memoria disponible en la máquina.

## 1.2. Estructura

Este trabajo está dividido en los siguientes capítulos:

1. Esta introducción, donde se plantea la motivación del presente trabajo, sus objetivos y la estructura de la memoria.
2. Una exposición del Estado del Arte en relación a los lenguajes de programación lógicos y la gestion de memoria así como conocimientos básicos.
3. Una pequeña descripción del lenguaje de programación Prolog y la lógica de $1^{er}$ orden sobre la que se fundamenta, con detalles sobre la implementación de Prolog en Ciao relevantes para este trabajo.
4. Descripción y funcionamiento del método de tabulación, así como de las estructuras de datos que maneja y su implementación en Tabling.
5. En el capítulo 5 se describe el trabajo realizado para resolver el objetivo 4 (Reinicialización ante la falta de memoria con Tabling activo). Implementación en Ciao, situación inicial en Tabling, implementación de la solución y trabajo futuro relacionado con este punto.
6. En el capítulo 6 se describe el trabajo realizado para resolver el objetivo 1 (Hacer compatible Tabling con la gestión de memoria en la WAM). Implementación en Ciao, situación inicial en Tabling, implementación de la solución y trabajo futuro relacionado con este punto.

7. En el capítulo 7 se describe el trabajo realizado para resolver los objetivos 2 y 3 (Gestionar las áreas de memoria de Tabling). Situación inicial en Tabling, implementaciones de dos soluciones y trabajo futuro relacionado con este punto.

8. Enumeración de las pruebas realizadas y los resultados obtenidos.

9. Las conclusiones y trabajo futuro más allá de resolver problemas de implementación.

Por último, el trabajo incluye: las referencias bibliograficas usadas para su elaboración y aquellas que pueden ser de utilidad para ampliar conceptos; y un anexo con el código que no se ha considerado relevante comentar en detalle.

# Capítulo 2

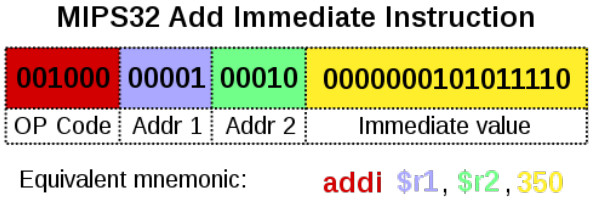
# ESTADO DEL ARTE

Los ordenadores y portatiles que utilizamos actualmente, son computadoras basadas en la arquitectura von Neumann, desarrollada en torno a 1950 por John Von Neumann(1903-1957). Los computadores basados en esta arquitectura tienen tres componentes principales: La unidad central de procesamiento[1] (CPU); la memoria y los módulos de entrada/salida (E/S). Estos componentes se comunican mediante un bus de datos. Sin embargo, la principal característica de este computador es que desarrolló la idea del programa almacenado. La idea del programa almacenado, término acuñado a partir de la publicación del "First Draft of a Report en the EDVAC" en 1945[2], escrito por von Neumann [12], planteaba la posibilidad de que los programas se almacenasen junto a los datos en una mismo memoria (otros modelos de computadores como los basados en la arquitectura Harvard, mantienen dos memorias, una para los programas y otra para los datos).

Los computadores con programas almacenados permitían utilizar un único computador para distintas tareas con solo cambiar el programa almacenado en memoria. Las instrucciones de estos programas estaban escritos en lenguaje máquina. Cada procesador es capaz de interpretar un determinado juego de instrucciones que determinan el lenguaje máquina de cada computador. Un programa es una sucesión de instrucciones a ejecutar por el computador. Estas instrucciones son una sucesión de 0's y 1's, que son la abstracción de los dos niveles de tensión de un sistema digital. Para acercar la programación de los computadores a un lenguaje inteligible por el ser humano, se desarrolló el lenguaje ensamblador, que consiste en una relación uno a uno con el código máquina, como se puede apreciar en la siguiente imagen que muestra la "traducción" de una intrucción que guarda en la dirección de memoria Addr1 el resultado de sumar el valor de la dirección de memoria Addr2 y 350.

[1] Que a su vez se divide en: unidad de control, banco de registros y unidad aritmético lógica (ALU).

[2] Con anterioridad J. Presper Eckert y John Mauchly (coautores del informe de von Neumann) en diciembre de 1943 y Alan Turing en 1936 ya habían tratado este concepto pero sin la repercusión del informe de von Neumann.




Las computadores entienden el lenguaje máquina, que si bien nos permite optimizar el rendimiento y el uso de memoria de la computadora, presenta dificultades de portabilidad (cada computadora responde a un determinado lenguaje máquina) y de programación. La aparición de lenguajes como fortran y C, que podemos catalogar como lenguaje de bajo nivel, resolvían estos problemas de portabilidad y facilitaban la programación mediante un sistema de símbolos y unas reglas sintácticas y semánticas.

A su vez los lenguajes de programación denominados de alto nivel, aumenta la facilidad con que nos comunicamos con los computadores y gracias a la evolución de los compiladores[3] no suponen una gran perdida de rendimiento frente a programar directamente en ensamblador.

Los lenguajes de programación de alto nivel son la consecuencia de la formalización de diferentes paradigmas de programación. Los paradigmas de programación, como la programación declarativa, son una propuesta tecnológica que trata de resolver un problema.

Como ya hemos comentado, uno de los objetivos de los lenguajes de programación de alto nivel es poder programar con un **lenguaje más cercano al problema** que permita facilitar: el aprendizaje del lenguaje; y la lectura del programa para su mantenimiento.

Según define Lloyd en [7] "la idea principal de la programación declarativa es que: un programa es una teoría (en una lógica adecuada) y la computación es la deducción a partir de dicha teoría"[4]. De hecho cuando la teoría utilizada es la lógica hablamos de programación lógica, de la que Prolog, basado en la lógica de primer orden, es uno de los principales representantes.

En el presente trabajo explicaremos los aspectos fundamentales de la lógica de primer orden como teoría que sustenta el leguaje de programación Prolog. Analizaremos los elementos del lenguaje así como métodos de resolución para obtener conocimiento a partir de unas premisas a las que denominaremos programa.

[3] El compilador [1], es un progrmaa informático que traduce un programa escrito en un lenguaje de programación concreto a: lenguaje máquina para ejecutar directamente en un computador; o a bytecode en caso de ejecución en Máquina Virtual.

[4] "The key idea of declarative programming is that: a program is a theory (in some suitable logic), and computation is deduction from the theory".

El método de resolución de Prolog se denomina SLD (Selective Linear Definite clause resolution). Se trata de un método: completo, garantiza la obtención de todas las respuestas; y correcto, las respuestas que obtiene son correctas.

Sin embargo mostraremos programas en los que el método de resolución SLD entra en un bucle infinito o genera de manera infinita un mismo conjunto de respuestas. Como alternativa para resolver estos problemas explicaremos los fundamentos del método de resolución OLDT que ha dado lugar a la programación lógica con tabulación. Con esta solución aumentamos la capacidad descriptiva del lenguaje de programación Prolog.

El otro objetivo de los lenguajes de programación de alto nivel es **la gestión de Memoria implicita**. Como ya comentamos anteriormente una de las partes fundamentales del computador es la memoria, donde se almacena el programa y los datos que queremos procesar. La memoria en los computadores es un recurso importante del computador porque tiene una capacidad limitada.

Lenguajes de programación de bajo nivel como C o ensamblador permiten al programador optimizar el uso de memoria al poder direccionar datos directamente a los registros del procesador (considerados el nivel 0 de memoria). Sin embargo para el propósito del presente trabajo nos quedaremos con la diferenciación de memoria principal (que representa los niveles de memoria físicos de los computadores y es el Hardware, el computador, el que se encarga de gestionar el movimiento de datos entre sus distintos niveles) y memoria secundaria (que en nuestro caso es el disco duro del computador). La evolución de los compiladores de C hacen que estos pasen a ser responsables de optimizar el uso de los registros y la caché del computador en el que estamos compilando el código. Sin embargo el lenguaje C sigue dejando en manos del programador optimizar el movimiento de los datos entre la memoria principal y el disco. Esta posibilidad sin embargo entraña riesgos dado que:

- Aumentar las posibilidades de error al incrementar la complejidad del codigo.
- El programador debe gestionar la memoria liberando los recursos cuando ya no son necesarios (si no los libera el programa puede no fallar pero pudemos llegar a agotar la memoria disponible en la máquina lo que provocarâ que el sistema operativo tuviese que “matar” el programa).

La memoria virtual, es un mecanismo gestionado por el sistema operativo y el hardware que ha permitido al programador abstraerse de los problemas de movimiento de los datos entre los distintos niveles de jerarquía de la memoria sin perder por ello rendimiento. El lenguaje C sigue permitiendo gestionar la ubicación de los datos en el mapa de memoria. Durante la ejecución de un proceso tenemos cuatro tipos de datos en el mapa memoria:

- El texto: es el código del programa en ejecución.

- Datos estáticos: Existen durante toda la vida del programa, tienen asociada una posición fija en el mapa de memoria.
- Datos dinámicos asociados a la ejecución de una función: Estos objetos de memoira (variables locales y argumentos de funciones) se crean con la invocacion de la función y se liberan cuando termina la llamada. Se almacenan en la pila (stack) del proceso. Es el sistema operativo el encargado de gestionarla.
- Datos dinámicos, su ubicación y tamaño se determina durante la ejecución del programa. El programador es el encargado de asignar un bloque de memoria a los datos cuando los crea (con la función $malloc(tamao)$) y debe liberar (con la función $free(puntero)$) el espacio de memoria cuando ya no son necesarios. Este área de memoria se denomina heap.

Para reducir los problemas que genera la gestión de la memoria, los lenguajes de alto nivel ofrecen gestión implicita de la memoria, de modo que el programador no tiene que preocuparse de asignar memoria al crear los datos. Es mediante el recolector de basura como los lenguajes de alto nivel determinan qué datos ya no son necesarios y pueden ser liberados. En algunos lenguajes el recolector de basura puede ser invocado por el programador. En el caso de Prolog es la WAM la encargada de gestionar la memoria y el mecanismo de recolección de basura, basado en la propuesta de Appleby y otros en [3] en 1988. Se trata de un mecanismo que:

- Aplica el algoritmo de recorrido implementado en dos fases: Una primera fase el la que marca los datos no referenciados y que pueden ser liberados; y una segunda fase en la que ordena los datos válidos
- Requiere que no se realizen peticiones de memoria. De hecho solo puede ser ejecutado en momentos concretos durante la ejecución de un programa.
- Realiza el compactado de las áreas de memoria. Cuando se libera memoria el mayor problema es la existencia de huecos liberados entre zonas de memoria usada, es lo que se denomina fragmentación y puede provocar que no se puedan asignar bloque de memoria aún cuando la suma de huecos lo permitiría.

Como ya hemos comentado una implementación de Prolog es **Ciao**, desarrolado principalmente por la Universidad Politécnica de Madrid (laboratorio CLIP) e IMDEA Software Institute. `http://ciao-lang.org/`. Ciao está basado en las primeras versiones de SICStus y en la Warren Abstract Machine (WAM, descrita en [13] y [2]). En un lenguaje de programación semi-interpretado (como Java), en el que el código se compila a un bitcode que se ejecuta en una máquina virtual. Esto permite que un mismo bitcode pueda ser ejecutado en diferentes computadores en los que tenemos instalada la WAM.

En `ciao-lang.org` están disponibles las últimas versiones de Ciao:

La versión actual de Ciao es la CiaoDE 1.14.2, liberada el 15 de agosto de 2011.
Existe una versión para desarrolladores, la CiaoDE 1.15-1781-g328b907, disponible desde el 12 de junio de 2013.

Para la realización del presente trabajo se ha utilizado el código fuente del repositorio de Ciao, de modo que en todo momento se ha trabajado con la última versión de Ciao. Para la actualización del código se ha utilizado la herramienta Git. Git es un software de control de versiones, diseñado por Linus Torvalds, que permite la colaboración en el desarrollo de software, permite mantener las versiones locales de cada desarrollador actualizadas y lleva un control de las modificaciones.

La principal característica de Ciao como lo describe Manuel Hermenegildo y otros en [6] es que proporciona al programador características útiles de diferentes paradigmas y estilos de programación, y que el uso de cada una de estas características (incluyendo la de Prolog) se puede activar y desactivar a voluntad para cada programa. El lenguaje está diseñado para ser extensible de una manera sencilla y modular.

Una de estas extensiones es el modulo de tabulación para Prolog denominado **Tabling** y desarrollado por Pablo Chico [4]. Se trata de una version beta y en la documentación del módulo se especifia que estos momentos no tiene implementada funcionalidades de Ciao como:

- Reasignacion de areas de memoria (Stack reallocation).
- Recogida de basura (Garbage collection).
- Cortes (Cuts).
- Programación paralela (Parallelism).

Por lo tanto al importar el módulo Tabling, el tamaño de los stacks se amplian para permitir la ejecución de los programas ya que con el tamaño inicial practicamente cualquier programa desbordaría la memoria.

El objetivo del presente trabajo es dotar al módulo de Tabling de un gestor de memoria de modo que las áreas de memoria se dimensionen según las necesidades de cada programa.

# Capítulo 3

# LOGICA DE PRIMER ORDEN

La lógica de primer orden, sobre la que se basan el lenguaje de programación lógico Prolog, es un sistema formal compuesto por un lenguaje formal (Q), sobre el que se definen axiomas y reglas de inferencia, que trata de representar la realidad y obtener conclusiones a partir de los hechos observados.

Para representar los hechos observados, el lenguaje formal (Q) consta de:

- Términos:
  - Constante: determinan un individuo concreto del mundo que tratamos de representar con el lenguaje.
  - Variable: representan un conjunto de individuos que no estan determinados a priori.
  - Funcion: define un individuo o conjunto de individuos que cumplen una propiedad. Es una funcion no interpretada, representa un individuo pero no sabemos que individuo es. La función se aplica sobre un determinado número de terminos (nombres o variables). Al número de términos en general $n \geq 1$ se le denomina aridad de la función. Una función de aridad 0 equivale a una constante.
- Predicado: expresa la relación entre un determnicado número de términos (nombres, variables y/o funciones). Al número de términos $n$ lo denominamos aridad del predicado.
- Conectores lógicos ($\wedge \vee \rightarrow \neg$), que significan respectivamente *y*, *o*, *implica*, *negación*.
- Cuantificadores ($\forall \exists$), que significan respectivamente *para todo*, *existe*.

Con este lenguaje podemos construir fórmulas bien formadas de Q que representan conocimiento que expresamos en lenguaje natural: "Juan es su abuelo si Juan es el padre de su padre o el padre de su madre"

$$\forall x\ \exists y\ [\text{abuelo(Juan, x)} \rightarrow \\ (\text{padre(Juan, y)} \wedge \text{padre(y, x)}) \vee \text{padre(Juan, madre(y))}]$$

donde:

- Constante = Juan
- Variables = x, y
- Función = madre/1
- Predicados = abuelo/2, padre/2

Estas formulas pueden ser:

- Satisfacibles: Existe una interpretación y una sustitución de sus variables que la hacen verdadera.
- Verdadera en una interpretación: Toda sustitución de sus variables la haven verdadera.
- Válida: en toda interpretación es verdadera.

Una interpretación consiste en:

- Un dominio o universo de la interpretación, compuesto por constantes y funciones, que representan los individuos del universo.
- Una asignación a cada constante o función de la fórmula de una constante o función del universo.
- Una asignación a cada predicado de una función que determine si el predicado es verdadero o falso.

Como ya hemos indicado, el objetivo de la lógica de primer orden es decudir conocimiento a partir de las premisas. Para ello uno de los objetivos es determinar una interpretación que haga satisfacible la fórmula. Sin embargo nos encontramos con dos dificultades: La variedad de conectores y cuantificadores que dificultan la manipulación de las fórmulas y la existencia de infinitas interpretaciones posibles.

Para facilitar la manipulación de las fórmulas, las transformamos a la Forma Normal de Skolem (FNS) en su forma clausular donde:

- Clausula: es una disyunción ($\wedge$) de términos.
- La Forma Clausular es la conjunción ($\vee$) de las cláusulas. Donde todas las variables están ligadas por el cuantificador $\forall$.

El anterior ejemplo en FNS es:

$\forall x\ \forall y$ [padre(Juan, y) $\wedge$ padre(y,x) $\rightarrow$ abuelo(Juan, x)]

$\forall x$ [padre(Juan, madre(x)) $\rightarrow$ abuelo(Juan, x)]

Para tratar con un número finito de interpretaciones restringimos el dominio al universo de Herbrand, que se define para cada fórmula y en el que:

- El dominio está formado por los términos resultantes de aplicar todas las funciones a todas las constantes. Sin embargo el dominio de Herbrand es inifito si la fórmula tiene simbolos de función.
- Las instancias básicas de una cláusula en FNS es el resultado de sustituir las variables libres de la cláusula por términos del universo de Herbrand. Si el dominio de Herbrand es infinito también lo será en número de instancias básicas.
- Una interpretación de Herbrand es el resultado de sustituir las variables de la cláusula por términos del universo de Herbrand.
- Para dar valor de verdad a una FNS hay que asignar valor de verdad a todas las instancias básicas ($\forall$) y a todas las cláusulas ($\wedge$).
- **Teorema**: "Una FNS es insatisfacible si y solo si es falsa para todas sus interpretaciones de Herbrand".

  > Para evaluar la insatisfacibilidad de una fórmula, tenemos que evaluar un número finito de interpretaciones. Si ninguna la evalúa como verdadera entonces es insatisfacible.
  > Si para alguna interpretación de Herbrand la fórmula es verdadera entonces la fórmula es satisfacible.

- **Teorema de Herbrand**: "Un conjunto de cláusulas es insatisfacible si y solo si existe un conjunto de instancias básicas de estas cláusulas que es insatisfacible"

Pese a que el universo de Herbrand no siempre reduce el número de interpretaciones a una cantidad finita. Sin embargo permite generar de manera incremental conjuntos de instancias básicas, que podemos evaluar por orden de manera que el teorema de Herbrand nos garantiza que si el conjunto de cláusulas era insatisfacible,

en un número finito pasos, encontramos un conjunto de instancias básicas que es insatisfacible.

Con esta formulación teórica Robison en 1965 [9] presentó el método de resolución por unificación en el que se deducen nuevas instancias básicas a partir de las existente. De modo que si se obtiene una contradicción (un literal y su negación) se concluye que el conjunto inicial es insatisfacible. El método es: correcto (si se deduce una contradicción, el conjunto es insatisfacible); y es completo (si el conjunto es insatisfacible aplicando la regla de resolución llegaremos a una contradicción). El algoritmo se basa en una única regla de inferencia sencilla, la regla de resolución: "Si sabemos que se cumple $P \vee Q$ y que se cumple $\neg P \vee R$ entonces se puede deducir que se cumplirá $Q \vee R$.

La aplicación del método de resolución no es determinista, permite diferentes caminos de resolución, pues existen varias combinaciones a la hora de por ejemplo seleccionar las cláusulas a las que aplicar la resolución. Una de las estrategias de resolución es el SLD (Selective Linear Definite clause resolution) que se aplica sobre Cláusulas de Horn.

Las Cláusulas de Horn son cláusulas en las que como másimo hay un literal afirmado:

$$\neg B_1 \ \vee \ \ldots \ \vee \ \neg B_2 \ \vee \ A$$

La interpretación de esta cláusula es aplicando la Ley de Morgan:

$$\neg(B_1 \ \wedge \ \ldots \ \wedge \ B_2) \vee A$$

Y aplicando la equivalencia de la conectiva $\rightarrow$.

$$B_1 \ \wedge \ \ldots \ \wedge \ B_2 \ \rightarrow \ A$$

Es decir nuestro ejemplo en clausulas de Horn sería:

$\neg$ padre(Juan, y) $\vee$ $\neg$ padre(y,x) $\vee$ abuelo(Juan, x)

$\neg$ padre(Juan, madre(x)) $\vee$ abuelo(Juan, x)

El SLD aplica un cálculo deductivo sumamente eficiente con el que demostrar la corrección de argumentos, que es de gran utilidad para la extracción de respuestas. Se trata de un método completo para la extracción de respuestas:

> Si tenemos un conjunto de cláusulas que representan los hechos del mundo que estamos representando y formulamos una pregunta, la pregunta tiene respuesta si a partir de los hechos podemos deducir la existencia de una sustitución a la pregunta.

Una forma de "guardar" las sustituciones es utilizando el literal $respuesta/n$ donde $n$ es el número de variables de las que queremos conocer la sustitución. Dado que podríamos querer que se cumpliese más de un predicado $P_1/m_!$, tendríamos que añadir la siguiente cláusula al conjunto de cláusulas:

$$respuesta(X_1, ...X_n) \vee \ \neg P_1(Y_1, ..., Y_{m_1}) \ \ldots \ \neg P_i(Z_1, ..., Z_{m_i}) \ \ldots$$

de modo que la pregunta tiene respuesta si existe una deducción de $respuesta/n$ a partir del nuevo conjunto de clausulas. La resolución nos dice si la pregunta tienen respuesta y en caso afirmativo nos da como respuesta la sustitución de las variables.

Volviendo a nuestro ejemplo, si definimos individuos del mundo que estamos representando, tendríamos las siguientes cláusulas:

$$\neg\ padre(Juan, y) \ \vee \ \neg\ padre(y, x) \ \vee \ abuelo(Juan, x)$$
$$\neg\ padre(Juan, madre(x)) \ \vee \ abuelo(Juan, x)$$
$$padre(Juan, Luis)$$
$$padre(Juan, madre(Sara))$$
$$padre(Juan, madre(Carlos))$$
$$padre(Luis, Miguel)$$

Y si queremos saber de quien es abuelo Juan añadiríamos la siguiente cláusula al conjunto:

$$respuesta(X) \vee \ \neg abuelo(Juan, X)$$

y obtendríamos los siguientes resultados:

$$respuesta(Miguel)$$
$$respuesta(Sara)$$
$$respuesta(Carlos)$$

Es decir: X=Miguel; X=Sara; X=Carlos, son sustituciones del dominio representado por el conjunto de cláusulas para las que están son satisfacibles.

Un ejemplo muy interesante para aplicar este sistema de extracción de respuestas es el problema "el mono y los plátanos"[1]:

[1]Problema planteado en la asignatura de Logica del grado de Matemáticas e Informática de la ETSIInf

Un mono quiere comer un plátano del racimo que cuelga del techo de una habitación. Aunque por sí mismo no alcanza, sí podría subirse a una silla que hay en la habitación para hacerlo

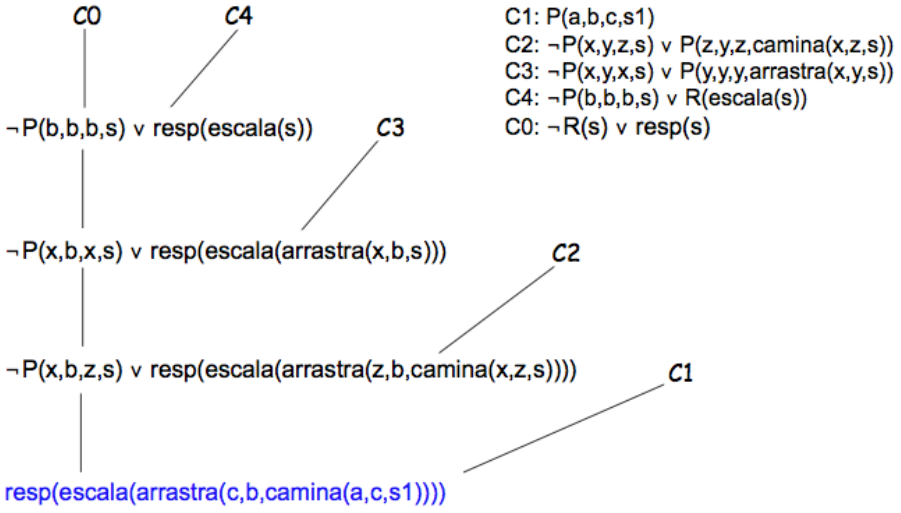
Lo primero que tenemos que hacer es definir nuestro lenguaje:

- Contantes:
  a - posición inicial del mono.
  b - p.i. del plátano.
  c - p.i, de la silla.
  s1 - estado inicial.
- Variables:
  x, y, z - posiciones posibles.
  s - estado posible.
- Funciones:
  camina(x,y,s) - estado alcanzado si el mono desde el estado $s$, camina de $x$ a $y$.
  arrastra(x,y,s) - estado alcanzado si el mono desde el estado $s$, arrastra la silla de $x$ a $y$.
  escala(s) - estado alcanzado si el mono desde el estado $s$ sube a la silla.
- Predicados:
  P(x,y,z,s) - en el estado $s$, el mono está en $x$, el plátano en $y$ y la silla en $z$.
  R(s) - en el estado $s$, el mono puede alcanzar el plátano.

La representación de la habitación del mono es:

- C1 - $P(a, b, c, s1)$ - en el estado inicial, el mono está en $a$, el plátano en $b$ y la silla en $c$.
- C2 - $\neg\ P(x, y, z, s)\ \vee\ P(z, y, z, camina(x, z, s))$ - desde cualquier estado $s$ el mono puede caminar desde donde está $x$ hasta la silla $z$.
- C3 - $\neg\ P(x, y, x, s)\ \vee\ P(y, y, y, arrastra(x, y, s))$ - desde cualquier estado $s$ el mono si está donde esta la silla $x$ puede arrastrarla hasta el plátano.
- C4 - $\neg\ P(b, b, b, s)\ \vee\ R(escala(s))$ - Si el mono y la silla están donde está el plátano $b$, el mono puede subirse a la silla para coger el plátano.
- La pregunta sería: C0 - $\neg\ R(s)\ \vee\ respuesta(s)$ - ¿Qué tiene que hacer el mono para poder coger el plátano?

El árbol de resolución al aplicar SLD sería:




Se puede observar como se ha aplicado el método de resolución. Partiendo de:

$$C0 = \neg\ R(s')\ \vee\ respuesta(s')$$

solo podemos aplicar la regla de resolución con:

$$C4 = \neg\ P(b,b,b,s)\ \vee\ R(escala(s))$$

de modo que tendríamos que realizar la sustitución $s' = escala(s)$ quedando el resolvente:

$$\neg\ P(b,b,b,s)\ \vee\ respuesta(escala(s))$$

. En la siguiente iteración solo podems resolver con C3 y así sucesivamente. De modo que obtenemos la respuesta:

$$respuesta(escala(arrastra(c,b,camina(a,c,s1))))$$

y que quiere decir que existe un estado en el que se cumple $R(s)$:

- El mono tiene que desde el estado $s1$ caminar de $a$ a $c$.
- Luego arrastrar la silla desde $c$ a $b$.
- Luego se sube a la silla para coger el plátano.

# Capítulo 4

# PROGRAMACION LOGICA

Una primera versión preliminar de Prolog para el procesamiento de lenguaje natural data de finales de 1971 por la Universidad de Marsella (Francia), bajo la direccion de A. Colmenauer utilizando resultados teóricos aportados por R. Kowalski (Universidad de Edimburgo). Los programas Prolog son programas lógicos definidos por cláusulas de Horn: hechos y/o reglas.

La traducción de la represntación de la familia de Juan que definimos en el capítulo anterior es el siguiente programa Prolog:

```
abuelo(juan, X) :-
        padre(juan, Y),
        padre(Y, X).
abuelo(juan, X) :-
        padre(juan, madre(X)).

padre(juan, luis).
padre(juan, madre(sara)).
padre(juan, madre(carlos)).
padre(luis, miguel).
```

y al realizar la consulta obtendríamos:

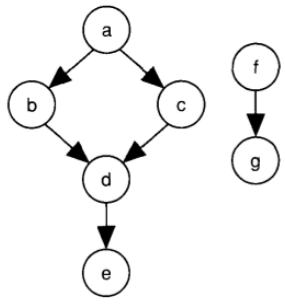
```
?- abuelo(juan, X).
X = miguel ? .
X = sara ? .
X = carlos ? .
no
?-
```

Como se puede observar la traducción desde el lenguaje de la lógica de primer orden al lenguaje de programación ha supuesto un cambio sintáctico pero no semántico. De hecho los términos que definimos al inicio siguen estando presentes y su definición no ha sufrido modificaciones. Sin embargo al referirnos a los elementos del programa lógico hay nuevas definiciones y restricciones que conviene conocer:

- Constante: Comienza con minúscula: $juan$, $sara$.
- Variable: Comienza con mayuscula: $X$, $Y$.
- Funtor: representa la funcion. Su nombre comienza con minuscula seguido de parentesis con los argumentos. El número de argumentos determina su aridad: $madre/1$.
- Hecho: es una cláusula de Horn que no tiene literales negados (tiene solo un literal afirmado): $padre(juan, luis)$.
- Regla: es una cláusula de Horn con un literal afirmado (cabeza) y una secuencia de literales negados (cuerpo). En el ejemplo $abuelo(juan, X)$ es la cabeza y $padre(juan, Y), padre(Y, X)$. el cuerpo.
- Procedimiento: un conjunto de reglas con la misma cabeza, se identifica con el nombre de la cabeza de las reglas y su aridad, en nuestro ejemplo $abuelo/2$.
- Programa Lógico: conjunto de procedimientos y hechos.
- Objetivo: es una cláusula de Horn sin literal afirmado. Equivale al cuerpo de una regla y correponde con la pregunta en la lógica de primer orden. $abuelo(juan, X)$.

## 4.1. Resolución SLD en Prolog

Sin embargo Prolog es mucho más que traducir lenguaje de primer orden. Un buen manual para iniciarse en la programación lógica es el libro "The Art of Prolog" [10] del que hemos sacado el siguiente ejemplo, que se representa un grafo diriguido con dos componentes conexas, para explicar como se realiza la ejecución de un programa en Prolog.




```
edge(a,b).
edge(a,c).
edge(b,d).
edge(c,d).
 edge(d,e).
edge(f,g).

reach(Node1, Node2) :-
        edge(Node1, Node2).
reach(Node1, Node2) :-
        edge(Node1, Link),
        reach(Link, Node2).
```

La imagen de la izquierda representa el grafo. A la derecha se puede ver la traducción en Prolog de dicho grafo. La primera parte corresponde a la definición del grafo a partir de las aristas diriguidas ($edge/2$) donde el primer argumento representa el

nodo origen y el segundo argumento el nodo destino de la arista. La segunda parte es un programa que define las reglas que tienen que cumplir dos nodos para considerar que están conectados.

- Desde un nodo 1 se llega a un nodo 2 si existe una arista desde el nodo 1 al nodo 2.
- Desde un nodo 1 se llega a un nodo 2 si existe una arista desde el nodo 1 a un nodo intermedio (*link*) y desde este nodo se llega al nodo 2.

Un programa Prolog se "ejecuta" realizando las preguntas adecuadas. Y al igual que en la lógica de primer orden obtendremos un modelo que hace ciertas las premisas. Prolog buscará las repsuestas y en caso de existir las mostrará. En nuestro caso podríamos estar interesados en conocer los nodos conectados desde el nodo $b$ de modo que al realizar la pregunta correpondiente el resultado sería:

```
?- reach(b,X).
X = d ? .
X = e ? .
no
?-
```

Pero una de las características de Prolog es que un mismo predicado (es como denominamos a los procedimientos) podemos realizar la cosulta al contraria sin modificar el código. Por ejemplo si queremos saber desde que nodos podemos llegar al nodo $d$:

```
?- reach(X,d).
X = b ? .
X = c ? .
X = a ? .
X = a ? .
no
?-
```

Se puede observar que el nodo $a$ aparece dos veces debido a que se puede llegar al nodo $d$ desde $a$ por dos "caminos".

El proceso de computación de Prolog que como ya hemos comentado se pone en marcha al realizar preguntas, esta basado en dos mecanismos:

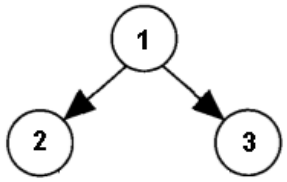
- Unificación, busca la instancia común más general de dos términos de Prolog.
- Resolución, unifica los objetivos de una pregunta con las clausulas obteniendo los resolventes. Se aplica de manera recursiva hasta que:
  - Se obtiene una clausula vacia, por lo tanto la resolución ha tenido exito y hay una respuesta a la pregunta realizada. En este caso se muestra la

respuesta y el usuario puede solicitar más respuestas (como hicimos en el ejemplo anterior) o puede deterner la ejecución.

- Si no se puede realizar la unificación, se realiza *backtracking* para evaluar otras clausulas alternativas. Cuando ya no hay más alternativas que explorar el programa devuelve *no*, indicando que no hay más respuestas.

El mecanismo de resolución utilizado se denomina SLD (Selective Linear Definite clause resolution), un mecanismo que recorre el arbol de busqueda de arriba a abajo y de izquierda a derecha, como pudimos ver en la resolución del problema de "El mono y los plátanos".

Para describir el recorrido de un árbol de resolución con varias alternativas en el que se ejecuta *backtracking*, vamos a reducir nuestro grafo a tres nodos:



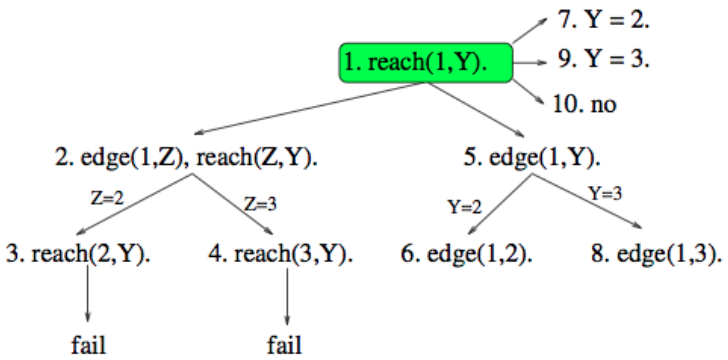

```
edge(1,2).
edge(1,3).

reach(X, Y) :-
        edge(X, Z),
        reach(Z, Y).
reach(X, Y) :-
        edge(X, Y).
```

el resultado de hacer la consulta de los nodos que se pueden alcanzar desde 1:

```
?- reach(1,X).
X = 2 ? .
X = 3 ? .
no
?-
```

Y el árbol de resolución correspondiente es:




En este caso no usamos el literal *respuesta* para guardar las sustituciones, estás se indican en los pasos de resolución. La clausula objetivo es $reach(1, Y)$ que tiene dos

alternativas de resolución. Al ejecutar SLD el segundo paso es evaluar la primera regla recorriendo el programa de arriba a abajo.

Una vez en el paso 2 del árbol buscamos las sustituciones que nos permiten unificar la primera de las cláusulas por la izquierda $edge(1, Z)$ que al buscar en el programa obtenemos como primera sustitución $Z = 2$.

En el paso 3 con $Z = 3$ nos queda una cláusula $reach(2, Y)$ que no podemos unificar con cláusulas del programa dando como resultado $fail$. Por lo tanto se ejecuta *backtracking* hasta el paso 2 donde tenemos otra alternativa para la sustitución $Z - 3$.

En el paso 4 con $Z = 4$ tambien fallamos y hacemos *backtracking* hasta el paso 1 donde hay otra alternativa a explorar que es $edge(1, Y)$.

En el paso 5 al evaluar $edge(1, Y)$, la primera sustitución posible es $Y = 2$.

En el paso 6 la sustitución $Y = 2$ se puede unificar con una cláusula del programa, quedándonos sin cláusulas para evaluar, por lo tanto $Y = 2$ es una respuesta del programa y se entrega en el paso 7.

El usuario en el paso 7 puede solicitar más respuestas o abortar la ejecución. Si continua se realiza *backtracking* desde el paso 6 hasta el paso 5 donde tenemos otra sustitución que explorar $Y = 3$.

En el paso 8 unificamos quedandonos sin cláusulas y por lo tanto entregamos la sustitución $Y = 3$ como respuesta en el paso 9. Si volvemos a solicitar más respuestas el programa hace *backtracking* hasta el paso 1 y como no hay más alternativas el programa falla y devuelve *no*.

## 4.2. Datos y áreas de memoria en Prolog

Sin entrar en detalles de como realiza Prolog la ejecución de un programa, a continuación voy a explicar algunas nociones de como se traducen los términos de un programa lógico a datos para que el interprete de Prolog ejecute el programa. Aunque de una implementación a otra de Prolog, los terminos son diferentes hay dos términos básicos:

- Variables: Su identificador empieza por mayuscula. Sirve para representar las variables
- Estructura: Su identrificador es el funtor que empieza por minuscula y su aridad $n$, un numero $n$ de términos $(f(t_1, ..., t_n)$. Representa las funciones y los predicados. Si la Estructura es de aridad 0 $(f/0)$, representa una constantes.

Estos términos se almacenan en un bloque de direcciones de memoria que denominamos **Heap**, también se denomina *global stack*, y que corresponde con un array de celdas de datos. La dirección de la celda es su indice en el Heap.

| | | |
|---|---|---|
| 0 | STR | 1 |
| 1 | $h/2$ | |
| 2 | REF | 2 |
| 3 | REF | 3 |
| 4 | STR | 5 |
| 5 | $f/1$ | |
| 6 | REF | 3 |
| 7 | STR | 8 |
| 8 | $p/3$ | |
| 9 | REF | 2 |
| 10 | STR | 1 |
| 11 | STR | 5 |

Las celdas están formadas por un tag (que determina de que término se trata) y una dirección de memoria.

- Variables <REF, k >, donde REF indica que es una variable y $k$ es la dirección de memoria:
  - Enlazada. La variable apunta a otra celda del Heap. Esto indica que son la misma variable porque se ha producido la unificación.
  - Sin enlazar. La variable se apunta a si misma.
- Estructura <STR, k >donde k apunta normalmente a la siguiente direccion de memoria (siguiente celda) donde se encuentra información sobre el funtor <f/n >. Las siguientes $n$ celdas del HEAP corresponden a los términos de los argumentos del funtor. Una estructura ocupa $n + 2$ celdas siendo $n$ la aridad del funtor.

La figura anterior es la representacion de $p(Z,\ h(Z,W),\ f(W))$ en el Heap. Como se puede observar en las celdas 10 y 11 la dirección a la que apuntan las estructuras no son las contiguas, eso es debido a que se tratan de términos de un funtor que como ya hemos indicado queremos que estén contiguos a partir de la celda que determina el funtor.

Otro de las áreas de memoria es el **Stack** donde se guardan los marcos de entorno. Los marcos de entorno surgen por la necesidad de preservar los resultados obtenidos

hasta un punto de la ejecución a partir del cuál el programa presenta dos posibles ejecuciones. Estos puntos con varias posibilidades se producen cuando un procedimiento tiene más de una regla. Estos puntos se denominan puntos de elección.

En la definición de la WAM realizada por David H. D. Warren en [13] y detallado posteriormente por Ait-Kaci en [2], los marcos de entorno y los marcos de elección se almacenan en un mismo stack con la intención de proteger los marcos de entorno ante un eventual *backtracking*.

Los marcos de entorno se generan cuando se trata de ejecutar una regla con un cuerpo de dos o más cláusulas pues determinar si existe solución y cada una de las cláusulas requiere la ejecución de la regla con la que dicha cláusula unifica.

Los puntos de elección se crean cuando en el árbol de resolución tenemos una bifurcación, es decir podemos unificar con varios términos o podemos resolver por varias reglas. El mecanismo de *backtracking* se basa en el análisis del árbol de resolución en profundidad de modo que cuando llegamos a un nodo que da fallo (no se puede unificar y no existen reglas de resolución) se deshace la ejecución realizada desde el último punto de elección y se sigue por el siguiente.

El mecanismo de *backtracking* también se activa cuando desde el top-level solicitamos más respuestas. Como hemos podido ver en el ejemplo anterior.

Como veremos a continuación, existe la posibilidad ya anticipada por Ait-Kaci en [2] de tener dos stacks diferenciados uno para los marcos de entorno y otro para los marcos de elección, refiriendose a ellos como AND-stack y OR-stack.

El último área de memoria dinámica es el **Trail**, esta área de memoria es la responsable de almacenar la información de las sustituciones realizadas durante la resolución. Guarda celdas de memoria como los descritas en el Heap.

Se trata de variables enlazadas condicionadas de las que el Trail tiene que recordar su referencia. Estas variables son aquellas creadas antes de un punto de elección, es decir la sustitución es validad siempre y cuando la resolución de resultado positivo. En caso contrario durante el *backtracking* hay que deshacer la sustitución para restituir el valor que tenían antes de iniciar la evaluación de dicha rama de resolución.

## 4.3. Datos y áreas de memoria en Ciao

La implementación de las celdas de memoria del Heap (y del Trail) en Ciao es la siguiente y los 32 bits de la variable se corresponden con:

```
typedef uint32_t tagged_t;
```

```
111 1 11111111111111111111111111 11
===== ========================== ==
tag   value                      GC
```

donde:

- Los tres primeros bits representan el *tag* que se refiere al tipo de término y como se puede observar disponemos de 8 terminos diferentes (en lugar de los dos términos básicos descritos anteriormente).
- El siguiente bit es el *subtag* que permite ampliar los tipos de términos.
- Los otro 28 bits son la dirección de memoria a la que apunta la celda.
- los ultimos dos bits son *GC* que se manipulan durante la recolección de basura para marcar las celdas que ya no son referenciadas y se pueden borrar.

La implementación en Ciao de los marcos de entorno es:

```
typedef struct frame_ frame_t;

struct frame_ {                    /* a.k.a. environment */
  frame_t *frame;                  /* continuation frame pointer */
  bcp_t next_insn;                 /* continuation program pointer */
  tagged_t term[FLEXIBLE_SIZE]; /* permanent variables */
};
```

Donde:

- frame = La dirección del siguiente marco de entorno (que referenciaremos al desalojar este marco de entorno).
- next_insn = Siguiente instrucción[1] a ejeuctar cuando se vuelve a activar el marco de entorno.
- term [] = Lista de tags que representan las variables locales de la regla, también llamadas permanentes y que corresponden a variables que aparecen en más de una clausula de la regla (la cabeza y la primera clausula del cuerpo se consideran una misma clausula)[2], es decir permanecen instanciadas durante la ejecución de varias clausulas.

La implementación en Ciao de los puntos de eleccion es:

[1] Las instrucciones están almacenadas en un área de memoria que no describimos al tratarse de un área de memoria independiente. Podemos decir que la WAM se asemeja a un computador en el que las instrucciones y los datos están en memorias independientes.

[2] En la siguiente regla A,B y C son temporales y el resto permanente:

$$p(A, B, X_1, X_2) : -q(A, B, C, C), r(X_1, X_2), r(X_2, X_1)$$

```
struct node_ {                    /* a.k.a. marker / choice point */
  tagged_t *trail_top;            /* trail pointer */
  tagged_t *global_top;           /* heap pointer */
  try_node_t *next_alt;           /* next clause at predicate entry */
  frame_t *frame;                 /* environment pointer */
  bcp_t next_insn;                /* continuation */
  frame_t *local_top;             /* local stack pointer, or NULL if invalid */
  tagged_t term[FLEXIBLE_SIZE]; /* temporary variables */
};
```

Donde:

- trail_top = La dirección a la que tiene que apuntar el último de las áreas de memoria que comentaremos a continuación (Trail) cuando volvemos a este punto del árbol de resolución.
- global_top = La dirección a la que tiene que apuntar el Heap cuando volvamos a este punto del árbol de resolución para tener las variables necesarias.
- next_alt = Siguiente clausula a evaluar si la rama de decisión previamente evaluada a fallado.
- frame = Marco de entorno a restituir a partir del cual reevaluar otra posibilidad.
- next_insn = Siguiente instrucción a ejecutar.
- local_top = La dirección del Stack
- term[] = Lista de tags que representan las variables locales de la regla, también llamadas permanentes.

Las áreas de memoria de la WAM se implementan en Ciao como pilas, estructura de datos en la que los datos se van amontonando uno encima de otro. Para saber donde hay que poner el siguiente dato se guarda la siguiente posición de memoria libre. Para referenciar las áreas de memoria se guarda la primera posición de memoria de la pila y para poder saber cuando se llena la pila se guarda la última posición válida del area de memoria. En el caso del Heap y el Stack existe otro puntero que cuando se sobrepasa activa el mecanismo de desbordamiento.

El Heap crece hacia direcciones de memoria crecientes. Las variables que definen sus límites son:

```
tagged_t *heap_start;           /* physical low bound */
tagged_t *heap_end;             /* physical high bound */
tagged_t *heap_warn;            /* initial heap overflow limit */
tagged_t *heap_warn_soft;       /* current heap overflow limit */
tagged_t *int_heap_warn;        /* Heap_Start if ^C was hit, else Heap_Warn */
```

El Stack, es el AND-stack según la propuesta de it-Kaci, crece hacia direcciones de memoria crecientes. Las variables que definen sus límites son:

```
tagged_t *stack_start;          /* physical low bound */
tagged_t *stack_end;            /* physical high bound */
tagged_t *stack_warn;           /* stack overflow limit */
```

El Choice-Stack (OR-stack) y el Trail comparten zona de memoria de modo que el Trail crece hacia direcciones de memoria creciente y el Choice-Stack crece hacia direcciones decrecientes de memoria. Las variables que definen sus límites son respectivamente:

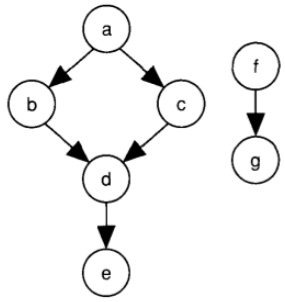
```
tagged_t *choice_start;         /* physical low bound */
tagged_t *choice_end;           /* physical high bound */

tagged_t *trail_start;          /* physical low bound */
tagged_t *trail_end;            /* physical high bound */
```

# Capítulo 5

# PROGRAMACION LOGICA CON TABULACION

Volviendo al ejemplo del grafo:




```
edge(a,b).
edge(a,c).
edge(b,d).
edge(c,d).
 edge(d,e).
edge(f,g).

reach(Node1, Node2) :-
        edge(Node1, Node2).
reach(Node1, Node2) :-
        edge(Node1, Link),
        reach(Link, Node2).
```

El mecanismo de resolución SLD que hemos explicado puede presentar problemas durante la ejecución de programas si los programas presentan pequeñas variaciones:

1. Realizan la recursión por la izquierda. Si en la definición del grafo anterior que realiza la recursión por la derecha invertimos la regla de recursividad:

```
reach(Node1, Node2) :-
        reach(Link, Node2),
        edge(Node1, Link).
```

   Y realizando la misma pregunta entramos en la ejecución de un bucle infinito.

```
?- reach(b,X).
X = d ? .
X = e ? .
own_realloc: could not reserve 134184960 chars!
{ERROR: Memory allocation failed [in Realloc()]}
{ Execution aborted }
?-
```

2. Base de hechos cíclica: Si en la base de hechos tenemos la representación de un grafo cíclico (no tiene que ser explicitamente un grafo como en el ejemplo, puede ser una relación entre individuos que genera un ciclo) como por ejemplo la arista del nodo $d$ al nodo $b$:

```
edge(d,b).
```

   Al realizar la misma pregunta el programa realizará la misma pregunta infinitas veces devolviendo las mismas preguntas repetidas.

```
?- reach(b,X).
X = d ? .
X = e ? .
X = b ? .
X = d ? .
X = e ? .
X = b ? .
X = d ? .
X = e ? .
...
```

## 5.1. Resolución OLDT en Prolog

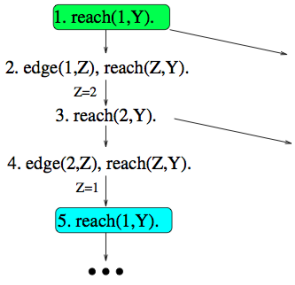
Podría parece que esto no es un problema porque hemos recibido las respuestas, pero si al igual que hicimos con la resolución SLD, simplificamos el grafo pero esta vez reordenamos las reglas:

```
edge(1,2).
edge(2,1).

reach(X, Y) :-
        edge(X, Z),
        reach(Z, Y).
reach(X, Y) :-
        edge(X, Y).
```

Al preguntar $reach(1, X)$, el programa entra en un bucle sin haber dado ninguna respuesta. Dado que su árbol de resolución es el siguiente

```
?- reach(1,X).
own_realloc: could not reserve 134184960 chars!
{ERROR: Memory allocation failed [in Realloc()]}
{ Execution aborted }
?-
```



Como se puede observar en el paso 5 se vuelve a aparecer la cláusula del paso 1 por lo tanto el resultado volverá a ser el mismo (en 5 pasos se volverá a realizar la misma consulta entrando en un bucle infinito).

Una solución a estos problemas es aplicar las ventajas de la resolución OLDT [11], al sistema de resolución SLD. El método de resolución OLDT, trata de evitar la computación reduntante recordando computaciones parciales y reusando las respuestas en llamadas posteriores.

El resultado de incorporar la tabulación al método de resolución de Prolog es:

- El aumento de la eficiencia al evitar recomputaciones de llamadas ya realizadas.
- La terminación de algunos programas Prolog que entran en bucles infinitos.

Para ilustrar el funcionamiento de la tabulación solo tenemos que incorporar el modulo Tabling y decalarar el predicado $reach/2$ como tabulado:


```
:- use_package(library(tabling)).
:- table reach/2.

edge(1,2).
edge(2,1).

reach(X, Y) :-
        edge(X, Z),
        reach(Z, Y).
reach(X, Y) :-
        edge(X, Y).
```

Al realizar la consultar $reach(1, X)$ obtenemos el siguiente resultado:

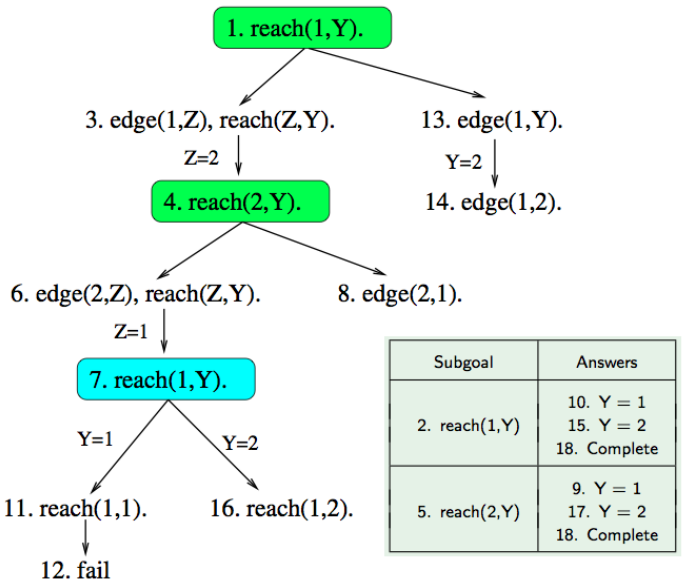
```
?- reach(1,X).
X = 2 ? .
X = 1 ? .
no
?-
```

Esto es gracias a que el compilador y el sistema $run - time$ guarda las llamadas del predicado tabulado de modo que:

- La primera llamada la guarda como *generador*.
- Cuando se repite la llamada, la detiene y la guarda como *consumidor*, continuando la ejecución con otra alternativa del árbol de resolución.
- Cuando se obtiene una solucion de la primera llamada esta se inserta en la tabla y se continua la búsqueda de soluciones.

- Una vez obtenidas todas las respuestas, se reactiva la ejecución de los consumidores que no tienen que recomputar la llamada pues obtienen las respuestas directamente de la tabla.
- Los predicados no marcados como *tabled* son ejecutados con la resolución SLD.

El árbol de resolución correspondiente es a la anterior ejecución es:




En el paso 1 se realiza la pregunta $reach(1, X)$, como se trata de un predicado tabulado, el sistema guarda en el paso 2 dicha cláusula como generador.

En el paso 3 hace una sustitución y tras realizar la unificación, en el paso 4 tiene que realizar una llamada a un predicado que de nuevo es el predicado tabulado por lo tanto en el paso 5 consultamos si está la llamada en la tabla y como no está, se guarda dicha llamada como otro generador (es importante observar que los generadores son distintos porque tienen constantes diferentes. Las variables de las cláusulas se renombran dado que las variables las cosideramos iguales).

En el paso 6 se realiza la sustitución y unificación por lo tanto llegamos al paso 7 en el que al detectar que es una llamada tabulada y está vez al consultar la tabla detectamos que es una llamada repetida y que por lo tanto es un consumidor. Se suspende la ejecución, se hace *backtracking* y se obtiene la primera solución para el generador $reach(2, Y)$. Se guarda la respuesta en el paso 9 en la tabla. Como la respuesta es también respuesta del primer generador, en el paso 10 se guarda la respuesta para dicho generador.

En el paso 11 se reactiva el consumidor suspendido para continuar la computación pero como la respuesta no aporta respuestas nuevas a los generadores se produce

fallo y se realiza *backtracking* hasta el paso 13 donde obtenemos otra respuesta para el primer generador $Y = 2$.

En el paso 15 guardamos la respuesta en la tabla y reactivamos el consumidor. El consumidor con la respuesta recibida aporta una solución al segundo generador que en el paso 17 guarda en la tabla.

El paso 18 es tras finalizar la ejecución del consumidor y realizar *backtracking* llegamos al paso 1 donde no hay más alternativas y se etiquetan los generadores como completos.

## 5.2. Algoritmo "Copy Hybrid Approach to Tabling"

Sin embargo la ejecución mediante tabulación tiene tres operaciones principales que no pueden ejecutarse todas ellas en tiempo constante:

- Suspender consumidores.
- Reactiva consumidores.
- Acceder a las respuestas.

La implementación en Ciao de la programación lógica con tabulación sigue el algoritmo CHAT (Copy Hybrid Approach to Tabling) [5], que no requiere cambios significativos ni en el compilador ni en el sistema run-time.

CHAT es un algoritmo basado en la suspensión de consumidores, su principal problema es el hecho de tener simultaneamiente varios árboles de resolución en el Stack que deben ser proteguidos frente a backtraking. Estos árboles de resolución tienen cada uno de ellos un conjunto de sustituciones diferente, que se conoce como multibinding.

La solución planteada en CHAT para resolver el problema de multibinding consiste en guardar las sustituciones condicionadas de las variables cuando suspendemos un consumidor y reisntalar dichas sustitucinoes al reactivarlos. Esta solución permite que los consumidores compartan el Heap y el Stack. Como consecuencia de esto la implementación según el método CHAT penaliza la operación de reactivar consumidores.

El otro concepto importante en esta solución es la congelación del Heap y el Stack. Con el objetivo de que los consumidores compartan el Heap y el Stack. Esta congelación hace que tras suspender la ejecución del consumidor el Heap y el Stack se mantienen y únicamente son desenlazadas las sustituciones (almacenadas en el Trail) desde el último punto de elección.

Para que en el momento de reactivar el consumidor, este tenga disponibles las sustituciones que han sido descartadas en el backtracking, se hace necesario almacenar dicha información en lo que llamaremos Chat Area del consumidor.

Sin embargo cuando tenemos llamadas tabuladas anidadas (esto es que se crea un generador $G_2$ antes de completar las llamadas de un generador anterior $G_1$) tenemos que tener especial cuidado al seleccionar las sustituciones que hay que guardar de los consumidores en especial cuando $G_2$ necesita de $G_1$ para completarse. En este escenario cuando un consumidor de $G_2$ suspenda y se realice backtracking hasta un punto de elección anterior a otro consumidor de $G_1$, cuando queramos reactivar el consumidor de $G_2$ este no tendrá disponibles las sustituciones que ha guardado el consumidor de $G_1$. Para resolver este escenario sin tener que duplicar la información guardando en los consumidores de $G_2$ las sustituciones de los consumidores de $G_1$ se crea un sistema de referencias en el que cada consumidor de $G_2$ tendrán un Chat Area enlazado con el Chat Area del consumidor de $G_1$ que se reactivó.

En cualquier caso determinar que sustituciones hay que guardar al llegar a un consumidor, se requiere simular el backtracking desde el punto de elección del consumidor hasta el punto de elección definido como generador. Esta simulación requiere consumo de computación y memoria por lo que según se describe [4] en Ciao se ha implementado un sistema que aplaza esta ejecución al momento en el que se realize el backtracking.

## Optimized CHAT

Esta solución, denominada Optimized CHAT (OCHAT), se basa en la idea de que al llegar a un punto de elección en el que se crea un consumidor, se marcan todos los puntos de elección existente entre este y el generador anterior de modo que al realizar backtracking sobre dichos puntos de elección se van guardando las sustituciones en el Chat Area del consumidor que marcó dichos puntos de elección.

Esta idea sin embargo no se puede aplicar directamente en Ciao porque Ciao descarta los puntos de elección cuando inicia la evaluación de la última de las alternativas posibles en un punto de elección. Para resolverlo, cuando se suspende la ejecución de un consumidor la dirección de memoria de la cima del Trail junto con un identificador del consumidor se guarda en el area de memoria del módulo de tabling (OCAHTStack). De modo que al realizar *backtraking* se ejecuta el *untrail*

- Si el elemento en la cima del OCHATSTack apunta a la cima del Trail:
  - Se inserta la sustitución en el CHAT area del consumidor asociado a la cima del OCHATStack.
  - Se decrementa la dirección a la que apunta la cima de OCHATSTack.

Si coincide con la dirección a la que apunta el elemento anterior del OCHATSTack, se puede quitar el elemento de la cima del OCHATStack.

  - Se deshace la sustitución y se vuelve a realizar el untrail

- Si el elemento en la cima del OCHATStack no apunta a la cima del Trail se deshace la sustitución y se vuelve a realizar el untrail

El untrail finaliza cuando se han descartado las sustituciones correspondientes a la rama de resolución sobre la que se está haciendo backtracking.

Cuando se completa un generador, las áreas del Heap y Stack que se habían congelado para compartir con sus consumidores no son necesarias de modo que se pueden recuperar. Esta operación es inmediata al restituir los punteros a las direcciones del Heap y el Stack anteriores a la creación del generador cuando se congelaron las posiciones del Heap y el Stack.

## 5.3. Datos y áreas de memoria en Tabling

La primera estructura que presentamos es evidentemente el generador:

```
struct gen
{
  long id;                          //generator identifier
  struct gen *leader;               //for precise SCC management.
  tagged_t on_exec;                 //free var if the generator is on execution.
  long state;                       //state of the call.
  struct gen *ptcp;                 //to represent Global Dependence Tree (GDT).

  struct sf* sf;                    //substitution factor of the generator.

  struct cons_list* first_cons;     //list of consumers.
  struct cons_list* last_cons;      //last consumer.

  trie_node_t *trie_ans;            //to check for repetitions.

  struct l_ans *first_ans;         //list of answers.
  struct l_ans *last_ans;          //last answer.

  frame_t *local_top;               //original stack freg.
  tagged_t *global_top;             //original heap freg.
  frame_t *stack_freg;              //original stack freg.
  tagged_t *heap_freg;              //original heap freg.
  tagged_t *tabl_stk_top;           //original tabling_stack_top
  node_tr_t *last_node_tr;          //original value of LastNodeTR.
  node_tr_t *cons_node_tr;          //NodeTR if gen -> cons

  node_t *node;                     //generator choice point
  struct cons_list *cons;           //pointer to consumer (non-leader generator)

  struct gen *prev;                 //Double generator linked list - prev
  struct gen *post;                 //Double generator linked list - post
};
```

Es la estructura que contiene practicamente toda la información del mecanismo de tabulación en Ciao. Como es puede observar guarda la siguiente información:

- id = Un identificador único del generador.
- leader = La dirección en memoria del generador leader[1].
- on_exec = Determina si el generador está en ejecución.
- state = Especifica el estado del generador.
- ptcp = La dirección en memoria del generador en la cima del ptcp. El ptcp es un stack que guarda de manera ordenada punteros a los generadores que se van creando (y descarta los que se van completando).
- sf = Estructura que permite hacer el siguiminto de las variables que son libres (no están enlazadas) en el momento de crear el generador. Se utiliza para insertar las respuestas obtenidas y para que los consumidores accedan a las respuestas.
- first_cons; last_cons = Dirección de memoria del primer y último consumidor.
- trie_ans = Dirección de memoria de la estructura de datos que almacena las respuestas de manera óptima para comprobar que la respuesta a insertar no está ya almacenada (para evitar repeticiones).
- first_ans; last_ans = Dirección de memoria de la primera y última respuesta.
- local_top; stack_freg = La dirección de memoria del Stack en el momento de crear el generador.
- global_top; heap_freg = La dirección de memoira del Heap en el momento de crear el generador.
- tabl_stk_top = Dirección de memoria de la cima de una de las áreas de memoria de Tabling (tabling_stack).
- last_node_tr = Valor del último nodo. Equivale a los elementos del OCHATStack.
- cons_node_tr = Elemento que el generador inserta en el OCATStack si finalmente es un consumidor.

[1]El proceso por el que se determina que un generador está completo, es decir, se han obtenido todas las respuestas posibles, es un proceso complejo especialmente si tenemos llamadas a generadores anidadas. Esta situación genera un Strongly Connected Component (SCC), que es representado por el leader, el último generador que no necesita de otros generadores en ejecución para ser completado. Este generador a su vez forma parte de otro SCC que será el siguiente en poder ser completado y así de manera recursiva.

- node = La dirección de memoria del punto de elección en el Choice-Stack.
- cons = Dirección de memoria de la lista de consumidores de un generador que no es leader.
- En este punto se han omitido las entradas que corresponden a una funcionalidad de Tabling no completamente implementada, denominada SWAPPING, que consiste en intercambiar los consumidores con los generadores con el fin de obtener respuesta de manera secuencial (como sucede con el SLD) en lugar de tener que esperar a completar la evaluación de los predicados tabulados.
- prev; post = Dirección de memoria del generador anterior y posterior.

La otra estructura determinante es evidentemente el consumidor:

```
struct cons
{
  struct sf* sf;                  //substitution factor of the generator.
  node_tr_t *node_tr;             //NodeTR
  frame_t *frame;                 //original stack freg.
  bcp_t next_insn;                //continuation program pointer
  struct l_ans *last_ans;         //last answer.
  struct gen *ptcp;               //to represent Global Dependence Tree (GDT).
  struct gen *gen;                //generator
  tagged_t ans_space;
  tagged_t attr_vars;
};
```

Como se puede observar: muchos de los atributos son identicos al generador por lo que no necesitan explicación; los dos últimos (*ans_space* y *attr_vars*) son estructuras de datos diseñadas para completar la funcionalidad de la tabulación con restricciones; por lo que solo comentaré (*next_insn*) que representa la siguiente cláusula a ejecutar cuando se reactive el consumidor.

Existen otras estructuras en Tabling como han podido observar en el codigo anterior que no comentaremos por no aportar conocimiento al objetivo del presente trabajo.

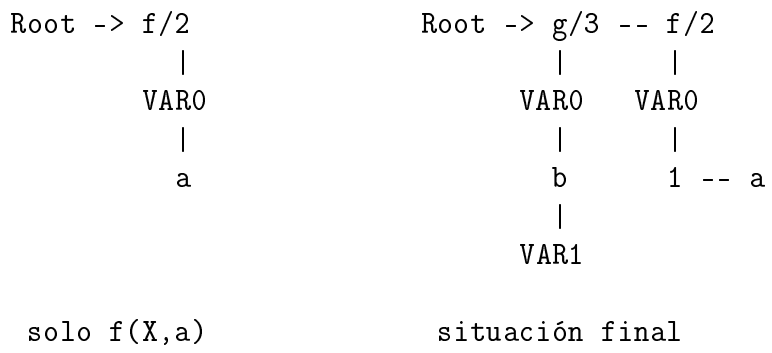
## El Trie

El trie es una estructura de datos y en la implementación de Tabling adquiere gran importancia porque la eficiencia en el acceso y manipulación de los datos tabulados es crítica. El término Trie proviene de *retrieval* en [8], recuperación de información. Se trata de una estructura de datos tipo árbol en el que los elementos que comparten sufijos comunes tienen un tronco del árbol común. Partiendo de la raiz, los elementos se construyen concatenando el valor de los nodos que recorre hasta llegar a la hoja.

Si quisiesemos insertar en un Trie los siguientes términos de Prolog:

- f(X,a).
- g(X,b,Y).
- f(Y,1).

debemos recordar como se descomponen los términos en Prolog: f(X,a) equivale a tres tags: una para el funtor f/2; otra para la variable X; y otra para la constante a. Dado que los generadores y consumidores se suponen equivalentes bajo renombrado de las variables, se realiza un renombrado de las mismas de modo que se denominan VAR0, VAR1, ... , VARn según aparecen en el término. Quedando los anteriores términos de la siguiente forma:




En la representación de la derecha cuando hemos insertado el primer término:

- Root es la dirección de memoria del último término añadido.
- Cuando solo hemos añadido f(X,a), tenemos tres nodos de modo que el nodo f/2 tiene un hijo que es VAR0 y este otro hijo que es a.
- Al añadir g(X,b,Y) se genera otro nodo que pasa a ser al que apunta Root. El nodo siguiente al nodo g/3 es el nodo f/2. El resto de hojas se construyen como en f/2.
- Al insertar f(Y,1), se recorre el árbol y se vé que f/2 ya está en el trie, Y se renombra a VAR0, que también está de modo que solo hay que crear el nodo 1 que pasa a ser el hijo de VAR01 y el nodo a que ya estaba pasa a ser el nodo siguiente al nodo 1.

Como se puede observar la representación no es exactamente un árbol, pero representa fielmente como se construye el Trie en Tabling.

La implementación de un nodo del Trie en Tabling es:

```
typedef struct trie_node_ trie_node_t;
typedef trie_node_t *TrNode;
struct trie_node_ {
  tagged_t entry;
  trie_node_t *parent;
  trie_node_t *child;
  trie_node_t *next;
};
```

Donde:

- entry = Es la representación del término tal y como hemos descrito en el cápitulo 3.
- parent; child; next = Los nodos padre, hijo y siguiente corresponden con las direcciones de memoria de los nodos del Trie tal y como hemos indicado anteriormente.

Los términos que guardamos en el Trie representan las llamadas (los generadores y consumidores) y las respuestas. Cuando tenemos una nueva llamada de una clausula tabulada se recorre el Trie para determinar si:

- Es generador (no existe una entrada que coincida). Se crea un nuevo nodo con sus correspondientes hijos tal y como hemos descrito anteriormente.
- Es consumidor de un generador que está en ejecución. Cuando llegamos a la hoja y nuestro término existe miramos el siguiente nodo y observamos que su valor es EXECUTING. Esto significa que el generador asociado está en ejecución y que no hay respuestas.
- Es consumidor de un generador que está completo. Idem pero el valor del siguiente nodo es COMPLETE. Esto significa que el generador asociado está completo y que se pueden extraer las respuestas.

Pero lo más destacable es como combina la estructura del trie con tablas hash cuando la profundidad del trie es mayor que un determinado valor (en Ciao se ha definido como límite de profundidad del trie 8 niveles).

Al igual que con los nodos del Trie, un nodo Hash es de gran simplicidad:

```
typedef struct trie_hash {
  tagged_t entry;  /* for compatibility with the trie_node data structure */
  int number_of_buckets;
  int number_of_nodes;
  trie_node_t **buckets;
  struct trie_hash *next;
  struct trie_hash *prev;
} *TrHash;
```

Donde:

- entry = Sirve para unificar estos nodos con los del Trie ante consultas en el recorrido del Trie.
- number_of_buckets = Indica el número de buckets disponibles para el Hash. Esta valor inicialmente es 64 y se incrementa al doble en cada ocasión que el Hash se queda sin espacio.
- number_of_nodes = Indica el número de nodos que se han insertado en el Hash. Cuando se completa el Hash, se solicita un nuevo espacio de memoria por el doble de espacio ocupado por los nodos.
- buckets = Es un array de direcciones de memoria de nodos del Trie que se ordenan como pequeños Trie en la tabla Hash.
- next; prev = Las tablas Hash están doblemente enlazadas con la dirección de la tabla Hash anterior y la posterior.

Las tablas Hash permiten acceder a los términos con mayor rápidez que en el trie a costa de perder la ventaja de compartir la parte común de los términos.

Es importate el análisis de los datos que definen el trie para poder realizar el recorrido de sus nodos en el procedimiento de ajuste de punteros que se ejecuta después del realojo de las áreas de memoria del modulo de tabling.

# Capítulo 6

# CAPTURA DE ERRORES

Los computadores durante la ejecución de un programa pueden producir errores que si no están previstos pueden derivar en instrucciones no admitidas por el computador. En estos casos el Sistema Operativo (SO) del computador que está atento a que los procesos no ejecuten instrucciones ilegales toma la decisión de matar el proceso que está ejecutando el programa.

Durante la ejecución de un programa el SO no está en ejecución por lo que se hace necesario que el computador a nivel de hardware sea el que informe al SO mediante interrupciones que algo raro está pasando. Como por ejemplo:

- Ante una instrucción ilegal como la división por cero. Cuando el procesador trata de ejeuctar dicha instrucción se genera una interrupción que el SO evalua y comunica al proceso. Si el proceso no está diseñado para poder resolver esta situación (por ejemplo informando al usuario de que la operación no es posible) el proceso deja de ejecutarse.
- Cuando el proceso requiere más espacio de memoria del que tiene asignado por el SO para trabajar. Como comentamos en el capítulo 2 los procesos disponen de dos åreas de memoria dinámicas (heap y stack) pero el SO va habilitando dichas áreas según crece la necesidad del proceso. Cuando el proceso necesita más stack para almacenar los datos de la ejecución de un procedimiento del que tiene asignado el computador lanza una excepción pues no puede atender la solicitud del proceso para guardar los datos en memria. En estos casos el SO analiza la interrupción y:
  - Si hay memoria física disponible en el computador para atender la solicitud del proceso aumenta el tamaño asignado al Stack y devuelve el proceso a ejecución.
  - Si el proceso ha agotado toda la memoria física disponible el SO tiene que matar el proceso.

En resumen, las interrupciones no siempre son acciones ilegales, en ocasiones son acciones que requieren la intervención del SO para poder continuar el programa.

Con esta protección del SO evitamos que un error en un programa suponga tener que reiniciar el computador. De hecho cuando es el SO el que realiza una acción ilegal el computador se apaga o se queda colgado y tenemos que reiniciar el ordenador.

## 6.1. Funcionamiento de la captura de errores en Ciao

Al igual que sucede con los computadores, la WAM que no deja de ser la abstracción de un computador, y Ciao tiene implementados unos mecanismos de control para impedir que operaciones de riesgo como solicitud de memoria o divisiones, puedan comprometan el funcionamiento del sistema.

El primer mecanismo consiste en ejecutar estas operaciones a través de una macro que captura errores de la siguiente manera:

```
#define CHECK_FOR_MEMORY_FAULT(ALLOC_CALL)  {          \
    tagged_t *__ptr = (ALLOC_CALL);                    \
    if (!__ptr) SERIOUS_FAULT(tryalloc_errstring);     \
    return __ptr;                                      \
  }
```

El ejemplo anterior muestra como se capturar el posible error cuando solicitamos al SO un espacio de memoria. Sabemos que existe la posibilidad de que ya no quede espacio de memoria físico para atender nuestra petición por lo tanto si el procedimiento que solicita dicho espacio de memoria no ha sido satisfactorio (en nuestro caso cuando esto sucede la dirección de memoria devuelta por el procedimiento ($__ptr$) es 0, lo cual equivale a decir que no exite dicha direccion de memoria, haciendo cierta la condición $if$ (!$__ptr$)...).

```
tagged_t *checkalloc(int size)
{
  CHECK_FOR_MEMORY_FAULT(tryalloc(size));
}
```

En resumen, cuando ejecutamos *checkalloc*(1000) el programa protege dicha ejecución de modo que si *tryalloc*(1000) al solicitar un espacio de memoria al SO no recibe una dirección de memoria donde se encuentran los 1000 Bytes solicitados la WAM ejecutará el procedimiento definido en $SERIOUS\_FAULT()$:

```
#define SERIOUS_FAULT(Y)        {failc(Y); \
                                 LONGJMP(abort_env, WAM_ABORT); }
```

La primera función $failc()$ se encarga de mostrar por la salida de errores definida por la WAM el mensaje $tryalloc\_errstring$ que se definió durante la ejecución del procedimiento (en nuestro ejemplo $tryalloc(1000)$).

La segunda función $LONGJMP()$ es una macro para abstraer los diferentes computadores en los que está ejecutandose la WAM. Como ya hemos indicado una maquina abstracta tiene como principal característica que en diferentes computadores con diferentes sentencias de SO ejecutan el mismo bytcode (código compilado). $LONGJMP$ en computadores con SO Unix es $longjmp(jmp\_bufenv, intval)$, una sentencia que restaura el entorno previamente guardado en el primer argumento en una invocación previa de $setjmp(jmp\_bufenv)$. El valor del segundo argumento de $longjmp$ es el valor que devolverá la función $setjmp$ cuando se restaure el entorno guardado y la computadora continue la ejecución con la siguiente sentencia después del $setjmp$.

En nuestro caso la ejecución sigue en el siguiente punto:

```
int ciao_firstgoal(ciao_state state, ciao_term goal) {
  int i, exit_code;
  worker_t *w;

  w = state->worker_registers;
  w->node->term[0] = X(0) = ciao_unref(state, goal); /* WAS A init_atom_check
      !!! (and goal was goal_name) */
  w->next_insn = bootcode;
  eng_killothers_ciao_prolog = eng_killothers_noop;

  while(TRUE) {
    i = SETJMP(abort_env);
    if (i == 0){                  /* Just made longjmp */
      w->term[0] = w->node->term[0];
      wam_initialized = TRUE;
      exit_code = wam(w, state);
      flush_output(w);
      if (exit_code != WAM_ABORT) /* halting... */
        break;
    }
    else if (i == -1) {           /* SIGINT during I/O */
      CIAO_REGISTER tagged_t *pt1;
      /* No need to patch "p" here, since we are not exiting wam() */
      CIAO_REGISTER bcp_t p = (bcp_t)int_address;
      int_address = NULL;
      SETUP_PENDING_CALL(address_true);
      continue;
    }
    reinitialize(w);                                    /* aborting... */
    init_each_time(w);                 /* Sets X(0) to point to bootcode */
    *(definition_t **)(bootcode+2) = address_restart;
  }
  return exit_code;
}
```

De modo que cuando se ejecuta $SERIOUS\_FAULT$ el programa vuelve a $i = SETJMP(abort\_env)$ y como la variable $i$ tomará el valor de $WAM\_ABORT$, distinto de 0 y -1, la siguiente sentencia que se ejecuta es $reinitialize(w)$ obteniendose como resultado una nueva ejecución de la WAM en la que las áreas de memoria

y otras condiciones de entorno están como al iniciarse la WAM por primera vez.

## 6.2. Implementación de la captura de errores en Tabling

Cuando se activa el módulo Tabling los mecanismos de tratamiento de fallos de la Wam siguen funcionando correctamente, sin embargo, no se ejecuta la reinicialización.

Paralelamente a este problema analizando el código de Tabling podemos observar que en las operaciones de *ALLOC_GLOBAL_TABLE* y *ALLOC_TABLING_STK* se comprueba si estamos dentro de los límites del área asignada y en caso contrario se escribe por la salida del proceso un mensaje con el error pero no se ejecuta ninguna operación adicional como se puede observar en el siguiente código:

```
#define ALLOC_GLOBAL_TABLE(PTR,PTR_TYPE,SIZE)                    \
  {                                                              \
    (PTR) = (PTR_TYPE)global_table_free;                         \
    global_table_free += (SIZE) / sizeof(tagged_t*);             \
    if (global_table_free >= global_table_end)                   \
      fprintf(stderr, "Global table memory exhausted\n");        \
  }                                                              \
```

Para mejorar el comportamiento de Ciao con el módulo Tabling realizaremos dos aportaciones: La primera es diseñar un procedimiento, al que hemos llamado *reinit_tabling*, que reinicie las estructuras del tabling; y luego activaremos el procedimiento de tratamiento de fallos antes de que la WAM se vea comprometida con una interrupción no capturada.

Para implementar *reinit_tabling* se hizo necesario resolver los siguiente problemas:

- Implementar una llamada al procedimiento de *reinit_tabling* que solo se ejecute cuando el módulo Tabling está activo.
- Acceder a un procedimiento del módulo tabling desde la WAM. Ciao compila el *engine* de manera independiente del módulo. Esto provoca que desde procedimientos de la WAM no tenemos conocimiento de las funciones del módulo porque todavía no han sido compiladas.
- Implementar el procedimiento que reinicia las estructuras de la WAM. En una primera implementación se repiten las acciones previstas en *initial_tabling_c*. Repetir el código da la posibilidad de adaptar como queremos que se reinicie el módulo.

- Crear casos de prueba para provocar una excepción de la WAM y evaluar el funcionamiento de la reinicialización.
- Resolver los fallos que surguen durante las pruebas de la implementación.

La primera tarea es sencilla, se determina que el momento de ejecutar *tabling_reinit* es una vez completada la función de *reinitialize* y para garantizar que solo se activará cuando queremos tener Tabling utilizamos el operador del preprocesador $\#ifdefined(TABLING)$ de modo que solo cuando hemos definido dicha variable el compilador compila esta parte del código:

```
/* Cleanup after abort: shrink stacks to initial sizes. */
CVOID__PROTO(reinitialize)
{
...
#if defined(TABLING)
  reinit_tabling(Arg);
#endif
}
```

Acceder a un procedimiento del módulo de Tabling presenta más dificultades. La solución implementada consiste en definir los procedimientos C del módulo Tabling como predicados Prolog. Para ello se declara el predicado y se dirigue su ejecución a un procedimiento en C:

```
:- true pred reinit_tabling + foreign_low(reinit_tabling_c).
```

luego hay que acceder al predicado desde la WAM. Esto implica que la operación tiene que buscar la dirección de memoria donde se encuentra guardo el procedimiento e indicar al computador que ejecute el procedimiento alojado en dicha dirección de memoria:

```
CVOID__PROTO(reinit_tabling) {
  tagged_t *junk, key;
  definition_t *d = NULL;

  key = MakeString("tabling_rt:reinit_tabling");
  d = find_definition(predicates_location, key, &junk, FALSE);
  if (d != NULL) d->code.cinfo(Arg);
}
```

solo cuando el usuario a activado el módulo de Tabling en su programa el *engine* encontrará el predicado y realizará la reinicialización del Tabling. Esto protege al sistema de realizar operaciones no previstas al no haber activado Tabling.

Finalmente hay que crear el procedimiento de *reinit_tabling_c*

```
/*
 * This function reinit tabling after reinitialize the wam
*/
CBOOL__PROTO(reinit_tabling_c)
{
```

```
#if defined(TABLING)
#if defined(DEBUG)
   if (tabling_debug == atom_on){
     printf("Reinit tabling\n");
     statistics_tabling_c(Arg);
   }
#endif

   DEALLOC_GLOBAL_TABLE;
   DEALLOC_TABLING_STK(tabling_stack);
   init_tries_module();
   trie_node_top = NULL;
   last_gen_list = NULL;
   tmp_term = (tagged_t)NULL;

#if defined(DEBUG)
   if (tabling_debug == atom_on)
     statistics_tabling_c(Arg);
#endif

   iptcp_stk = 0;
   ptcp_stk = (struct gen**) checkalloc (PTCP_STKSIZE * sizeof(struct gen*));
   PUSH_PTCP(NULL);

   INIT_NODE_TR(initial_node_tr);
   INIT_NODE_TR(LastNodeTR);
   address_nd_consume_answer_c = def_retry_c(nd_consume_answer_c,3);
   address_nd_consume_answer_attr_c = def_retry_c(nd_consume_answer_attr_c,5);
   address_nd_resume_cons_c = def_retry_c(nd_resume_cons_c,3);
   address_nd_back_answer_c = def_retry_c(nd_back_answer_c,1);

#if defined(DEBUG)
   if (tabling_debug == atom_on)
     statistics_tabling_c(Arg);
#endif

   return TRUE;

#else
   printf("\nTABLING Flag must be activated\n");
   return FALSE;
#endif
}
```

Para realizar las pruebas se decidió provocar un error cuando la WAM se dispone a ampliar sus áreas de memoria. Para ello localizamos el lugar donde se inicia el procedimiento de *overflow* y creamos un mecanismo que mediante flags nos permitia activar o desactivar la ejecución de la excepción. Esta decisión nos permitió: evaluar el uso de flags previsto en la WAM; evaluar el mecanismo de gestión de la necesidad de incrementar memoria en la WAM.

Cuando finalmente la implementación se ejecutaba correctamente pudimos observar que la WAM no era capaz de restarurar el funcionamiento del top-level. El análisis de la traza de la implementación realizada no mostraba problemas lo que hizo necesario realizar un análisis con el depurador de C para OS X (llgb). Analizando la traza generada por el depurador concluimos que se producia un comportamiento diferente en la ejecución de la wam cuando se reinicializaba sin Tabling y con Tabling. Dicha diferenica se producía en *wam.c* : 152:

```
if (desc && (desc->action & BACKTRACKING)) {
```

donde al evaluar el valor de $goal\_desc->action$ en lugar de $NO\_ACTION$ tenia el valor de $BACKTRACKING$. Ante la dificultad de determinar porque se producia esta diferencia procedimos a colocar un parche antes de la llamada que activa la WAM:

```
int ciao_firstgoal(ciao_state state, ciao_term goal) {
...
    if (i == 0){                /* Just made longjmp */
      Arg->term[0] = Arg->node->term[0];
      wam_initialized = TRUE;
      // MCL, JA: patch to avoid forced backtracking -- need to find out where
          ->action is changed
      goal_desc->action = NO_ACTION;
      exit_code = wam(Arg, goal_desc);
```

Esta solución provisional nos permitió obtener el funcionamiento deseado.

La segunda medida que implementamos para mejorar el control de fallos una vez implementada la función de reinit consistia en hacer que la WAM y el módulo se reiniciasen ante un desbordamiento de las áreas de memoria del módulo. Para ello gracias al conocimiento adquirido se concreta en introducir el procedimiento $SERIOUS\_FAULT$ en las siguientes macros del módulo:

```
#define ALLOC_GLOBAL_TABLE(PTR,PTR_TYPE,SIZE)                              \
  {                                                                        \
    (PTR) = (PTR_TYPE)global_table_free;                                   \
    global_table_free += (SIZE) / sizeof(tagged_t*);                       \
    if (global_table_free >= global_table_end) {                           \
      fprintf(stderr, "Global table memory exhausted\n");                  \
      SERIOUS_FAULT(" Trying ALLOC_GLOBAL_TABLE");                         \
    }                                                                      \
  }

#define ALLOC_TABLING_STK(PTR,PTR_TYPE,SIZE)                               \
  {                                                                        \
    (PTR) = (PTR_TYPE)tabling_stack_free;                                  \
    tabling_stack_free += (SIZE) / sizeof(tagged_t*);                      \
    if (tabling_stack_free >= tabling_stack_end) {                         \
      fprintf(stderr, "Tabling stack exhausted\n");                        \
      SERIOUS_FAULT(" Trying ALLOC_TABLING_STK");                          \
    }                                                                      \
  }
```

Como se puede observar se mantiene el mensaje de información aún cuando el procedimiento de tratamiento de fallos nos enviará el mensaje determinado en la llamada correspondiente.

# Capítulo 7

# MANEJO DE MEMORIA

Determinar una cantidad fija de memoria que debemos reservar para las áreas de memoria de la WAM independientemente del programa que se va a ejecutar tendría como consecuencia:

- Si realizamos una asignación generosa para poder ejecutar programas que requieren mucha memoria, estaríamos sobredimensionando los requisitos de memoria durante la ejecución de programa que requieren menos memoria.
- Dado que no todos los programas tienen las mismas necesidades de las diferentes áreas de memoria, se daría la circunstancia de que un programa agotase la memoria de una de las áreas de memoria y no pudiese continuar la ejecución teniendo memoria disponible en otra de las áreas.

Por lo tanto al igual que sucede con el SO cuando pone en ejecución un proceso, en la WAM las áreas de memoria son asignadas de manera dinámica. Esto significa que durante la ejecución de los programas, existe un mecanismo que atiende las necesidades de memoria de cada una de las áreas de memoria de la WAM.

Este mecanismo está implementado mediante un juego de sentencias propias de Ciao que gestiona una reserva de memoria global de manera independiente al SO. Esta reserva de memoria que se solicita al SO está dimensionada basandose en la capacidad de las estructuras de la WAM para determinar direcciones de memoria.

En un computador el número máximo de direcciones de memoria que se pueden determinar viene determinado por la longitud de la palabra del computador. La palabra en un computador es la cadena finita de bits que pueden soportar los registro de la CPU. De hecho es el tamaño de la palabra de un computador lo que determina que la arquitectura del computador sea de 32 o de 64 bits y en consecuencia el SO del computador será de 32 o 64 bits respectivamente. Como ya hemos comentado un computador es binario, entiende entre 0's y 1's de modo que con palabras de

32 bits puede generar $2^{32}$ palabras diferentes. Es pues esta cifra la que determina el tamaño máximo de unidad de memoria direccionable por un computador. La unidad de memoria es 8 bits (1 byte) de modo que la capacidad máxima de la memoria que podemos direccionar con 32 bits es:

$$2^{32} \text{ bytes} = 4{,}294{,}967{,}295 \text{ bytes} = 4 \text{ gigibytes}^1 \approx 4 \text{ gigabytes}$$

El hecho de que la CPU pueda direccionar esta cantidad de memoria no implica que exista físicamente. La cantidad de memoria física está limitada por: el ancho del bus de datos (el ancho de banda especifica el tamaño máximo en bits de los datos que puede transmitir el bus de modo que al igual que con la memoria solo podríamos acceder a $2^{ancho}$ bytes de memoria); la memoria RAM instalada en el computador.

## 7.1. Manejo de las áreas de memoria de Ciao

Haciendo un simil con los computadores, en Ciao, las palabras son los *tagged_t* ya definidos en el capítulo 3. Los *tagged_t* tienen un tamaño 32 bits de los que: 28 eran para la dirección de memoria. Esto implica que esta implementación de la WAM solo puede direccionar $2^{28}$ posiciones de memoria y esta cifra determina el tamaño máximo a solicitar al SO pues aún pudiendo disponer de más memoria no podríamos referirnos a ella. El tamaño solicitado está especificado en una variable ($\#define\ AddressableSpace\ 0x10000000$)[2] que equivale a:

$$1 * 16^7\ bytes = 268{,}435{,}456\ bytes = 2^{28}\ bytes = 256MiB$$

Una vez la WAM dispone de esta zona de memoria las necesidades de memoria para las distintas áreas de la WAM se gestionan mediante procedimientos definidos en Ciao. Para realizar la gestión de las zonas de memoria el sistema tienen un registro de los bloques de memoria con su ubicación, su tamaño, si están libres y cual es el anterior y posterior bloque de memoria. Los procedimientos en Ciao son:

```
#if defined(USE_OWN_MALLOC)
#  define Malloc(p)      own_malloc(p)
#  define Free(p)        own_free(p)
#  define Realloc(p, s) own_realloc(p, s)
tagged_t *own_malloc(int size);
tagged_t *own_realloc(tagged_t *ptr, int size_in_chars);
void own_free(tagged_t *ptr);
void init_mm(void);
#else
#  define Malloc(p)      malloc(p)
```

[1] En el Sistema Internacional: 1 kilobyte(kB), 1 megabyte (MB) y 1 gigabyte (GB) son respectivamente $10^3$, $10^6$ y $10^9$ bytes. Sin embargo cuando utilizamos el sistema binario nos referimos a $2^{10}$, $2^{20}$ y $2^{30}$ como kibibyte(KiB), mebibyte (MiB) y gibibyte (GiB).

[2] Número expresado en hexadecimal, o base 16, es la representación utilizada por los computadores para indicar las direcciones de memoria. Para expresar los 16 digitos de la base se utilizan 0-9, A, B, C, D, F, de modo que $0x3FA = 3 * 16^2 + 15 * 16^1 + 10 * 16^0 = 1018$

```
#  define Free(p)       free((char *)p)
#  define Realloc(p, s) realloc((char *)p, s)
#endif
```

como se puede observar existe la posibilidad de utilizar las funciones del SO. A continuación explico las particularidades del mecanismo de gestión de Ciao.

- Malloc(p): Busca un bloque de memoria libre adecuado para el tamaño solicitado $p$. Si no lo encuentra, crea un nuevo bloque, y devuelve la posición de memoria de la primera palabra del bloque de memoria.
- Free(p): En este caso $p$ representa una posición de memoria, y corresponde a la primera palabra del bloque de memoria. El sistema busca el bloque al que corresponde esta posición de memoria y cambia su estado a libre. Para reducir la fragmentación del disco en bloques pequeños, el sistema comprueba si el bloque anterior o posterior están libre y en caso afirmativo crea uno mayor que agrupa aquellos que están libres.
- Realloc(p,s): $p$ representa la posición de memoria del bloque que queremos realojar y $s$ la nueva dimensión que queremos que tenga el bloque de memoria. Para ello el sistema realiza un Malloc(s) y un Free(p), y devuelbe la posición de memoria obtenida al realizar Malloc(s).

En caso de que la operacion Malloc o Realloc no se pueda completar con exito porque no exista espacio disponible para satisfacer la solicitud de memoria, el procedimiento devuelve la posición de memoria 0x00 de modo que es el procedimiento que ha realizado la solicitud la que tiene que comprobar que la posición de memoria obtenida es correcta.

Ciao también lleva un registro de las áreas de memoria de la WAM que se puede consultar mediante el predicado *statistics*/0 y que ofrece la siguiente información:

```
memory used (total)       8525338 bytes
   program space (including reserved for atoms): 6562498 bytes
   number of atoms and functor/predicate names: 11344
   number of predicate definitions: 4994
   global stack       531220 bytes:     23596 in use,    507624 free
   local stack         16380 bytes:       352 in use,     16028 free
   trail stack         17030 bytes:      1968 in use,     15062 free
   control stack       15730 bytes:       668 in use,     15062 free

   0.000195 sec. for 2 global, 0 local, and 0 control space overflows
   0.000000 sec. for 0 garbage collections which collected 0 bytes

 runtime:      1.043947 sec.      1043947 ticks at      1000000 Hz
 usertime:     1.044142 sec.      1044142 ticks at      1000000 Hz
 systemtime:   0.079084 sec.        79084 ticks at      1000000 Hz
 walltime:     5.583984 sec.      5583984 ticks at      1000000 Hz
```

donde podemos consultar:

- Los bytes de memoria en uso.
- El tamaño de las áreas de memoria de la WAM (Heap = global stack; Stack = local stack; Trail = trail stack y Choice-Stack = control stack) así como bytes en uso y bytes libres.
- También lleva un registro del número de *overflows* que ha ejecutado en el Heap, Stack y en el Trail+Choice..

Durante el desarrollo del presente trabajo se hizo necesario conocer la dirección de memoria del inicio y el final de los stacks asi como el % de espacio ocupado en los stacks por lo cree una función auxiliar (*statistics_resume*()[3] que mostrase la siguiente información:

```
Heap   (1872482684 - 1873013904) in use    23596 of     531220 (4.44 %)
Stack  (1878824776 - 1878841156) in use      352 of      16380 (2.15 %)
Trail  (1878791988 - 1878824748) in use     1968 of      32760 (6.01 %)
Choice (1878824748 - 1878791988) in use      668 of      32760 (2.04 %)
```

para facilitar el acceso a la información sin sobrecargar Ciao con nuevos predicados, completé el procedimiento con el siguiente código:

```
CBOOL__PROTO(statistics)
{
...
#if defined(TABLING)
  if (tabling_debug == atom_on)
    statistics_resume(Arg);
#endif

  return TRUE;
}
```

A su vez cabe destacar que al iniciarse la WAM la dimensión de los stacks es la definida en las siguientes constantes:

```
# define kCells   1024

# define GLOBALSTKSIZE   (16*kCells-1) /* Was 6*kCells-1 (DCG) */
# define LOCALSTKSIZE    (4*kCells-1)
# define CHOICESTKSIZE   (4*kCells-1)
# define TRAILSTKSIZE    (4*kCells-1)
```

Si analizamos los tamaños de los stacks con relación a los valores aquí definidos observaremos que hay una relación 1:4 por ejemplo el Stack se define con un tamaño

$$4 * kCells - 1 = 16383 \text{ direcciones de memoria}$$

pero si miramos el tamaño del stack en bytes observamos que ocupa 16380. Esta diferencia se debe a que en LOCALSTKSIZE especificamos cuantas direcciones de

[3]El código de la función *statistics_resume*() está en los Anexos

memoria queremos tener. Dado que la unidad de medida de la WAM es *tagged_t* y su tamaño son 32 bits (4 Bytes) el tamaño inicial del Stack es de 16380 $Bytes = 16\ KiB$. En resumen cuando iniciamos la WAM el área de memoria destinada para los stacks es de $[(16 + 4 + 4 + 4) * kCells] * 4\ Bytes = 112\ KiB$.

## 7.2. Manejo de las áreas de memoria de Ciao con Tabling

El modulo de Tabling sin embargo no está diseñado para adaptarse al proceso de overflow que se produce cuando uno de los stacks se queda sin memoria.

Como ya hemos indicado Ciao cuando uno de los stacks se queda sin memoria solicita un nuevo bloque de memoria donde copia los datos y luego libera el bloque. Esta operación implica que los datos ya no estån en la dirección de memoria inicial de modo que los *tagged_t* que referenciaban datos alojados en una determinada posición e memoria ya no están y hay que modificar dicha dirección con la nueva posición.

Dada la complejidad de actualizar las direcciones de memoria de las estructuras de la WAM ante un overflow esta operación solo se ejecuta en momentos concretos durante la ejecución del algoritmo de resolución. La situación más delicada se produce ante falta de memoria del Heap o del Stack. No siempre es posible detener el algoritmo para realizar el overflow de estos stacks por lo que se ha determinado una zona de seguridad (mediante *heap_warn* y *stack_warn* ya comentado en el capítulo 3 de modo que cuando el Heap o el Stack sobrepasan dicha dirección de memoria se indica a la WAM que en la siguiente punto que permita ejecutar overflow se ejecute. Esta zona de seguridad ha sido determinada calculando el espacio máximo que los stacks pueden llegar a requerir desde el paso por dos puntos que permiten ejecutar el overflow.

Dado que la implementación del módulo tabling no actualiza los atributos de los datos que apuntan a las posiciones de memoria de los stakcs de la WAM la solución implementada es la de ampliar dichos stacks con las siguientes dimensiones:

```
#define TABLING_GLOBALSTKSIZE   (4800*kCells-1)
#define TABLING_LOCALSTKSIZE    (3000*kCells-1)
#define TABLING_CHOICESTKSIZE   (3000*kCells-1)
#define TABLING_TRAILSTKSIZE    (3000*kCells-1)
```

De modo que hemos pasado a $(4800+3000+3000+3000)*kCells*4 = 56524800\ B \approx 54\ MiB$). Anteriormente ya hemos comentado los inconvenientes de determinar la dimensión de los stack de manera estática. Uno de los objetivos del presente trabajo es hacer compatible el overflow de la WAM con el Tabling.

Se definen tres funciones en el *engine* de Ciao desde las que se localizan y ejecutan las funciones correspondientes en el modulo tabling. Las funciones son:

```
CVOID__PROTO_N(heap_overflow_adjust_tabling, int heap_factor);
CVOID__PROTO_N(stack_overflow_adjust_tabling, int stack_factor);
CVOID__PROTO_N(stack_overflow_adjust_tabling, int stack_factor);
```

El codigo de cada uno de ellos es semejante al implementado para *reinit_tabling*, con la particularidad que en este caso hay que indicar un argumento (*x_facor*) que determina en bytes la diferencia entre las variables que apuntan a la primera posición de memoria de los bloques de memoria antes y después de haberse ejecutado el realojo de memoria. El codigo de uno de ellos es:

```
CVOID__PROTO_N(heap_overflow_adjust_tabling, int heap_factor) {
  tagged_t *junk, key;
  definition_t *d = NULL;

  key = MakeString("tabling_rt:heap_overflow_adjust_tabling");
  d = find_definition(predicates_location, key, &junk, FALSE);
  if (d != NULL) d->code.cinfo(Arg, heap_factor);
}
```

De igual modo hay que definir los predicados en el modulo tabling (ver solución de reinit_tabling), e incorporar las llamadas a estas funciones desde el lugar adecuado en la WAM. Se muestra el código de la llamada a una de las funciones que se encuentra en el función de overflow invocada por la WAM una vez realizado el realojo de la memoria y tras calcular la diferencia entre las posiciones de memoria donde empiezan los bloques de memoria (*reloc_factor*):

```
#if defined(TABLING)
#if defined(DEBUG)
    if (tabling_debug == atom_on)
      printf("Llamando␣heap_overflow_adjust_tabling␣desde␣heap_overflow\n");
#endif
    heap_overflow_adjust_tabling(Arg, reloc_factor); // XS
#endif
  }
```

En el modulo tabling se implementa el codigo que ajusta aquellos datos de las áreas de memoria del módulo y que apuntan al área de memoria que ha sufrido overflow y ha tenido que realojarse.

- Ante un overflow del Heap: Recorremos todos los generadores y ajustamos el valor de los atributos que guardan el valor de la cima del Heap cuando se creó el generador. El código[4] que se ejecuta es:

```
/*
 * This function adjusts the pointers after overflow of wam's stacks
 */
CBOOL__PROTO_N(heap_overflow_adjust_tabling_c, int heap_factor)
```

[4]Se ha omitido el código de depuración y traza.

```
{
  if (heap_factor == 0)
    return TRUE;

  struct gen* aux = last_gen_list;

  while (aux) {
    if (aux->global_top)
      aux->global_top = (struct tagged_t*)((char *)(aux->global_top) +
          heap_factor);
    if (aux->heap_freg)
      aux->heap_freg = (struct tagged_t*)((char *)(aux->heap_freg) +
          heap_factor);
    aux = aux->prev;
  }

  return TRUE;
}
```

- Ante un overflow del Stack: Recorremos todos los generadores y ajustamos el valor de los atributos que guardan el valor de la cima del Stack cuando se creó el generador. Pero en este caso también tenemos que recorrer (si existen) la lista de consumidores asociados al consumidor porque también ellos han guardado el valor de la cima del Stack cuanado fueron creados. El código que se ejecuta es:

```
CBOOL__PROTO_N(stack_overflow_adjust_tabling_c, int stack_factor)
{
  if (stack_factor == 0)
    return TRUE;

  struct cons_list* cons_list;
  struct gen* aux = last_gen_list;

  while (aux) {
    cons_list = aux->first_cons;
    if (aux->local_top)
      aux->local_top = (frame_t*)((char)(aux->local_top) + stack_factor);
    if (aux->stack_freg)
      aux->stack_freg = (frame_t*)((char)(aux->stack_freg) + stack_factor);
    while(cons_list) {
      if (cons_list->cons->frame)
        cons_list->cons->frame = (frame_t*)((char)(cons_list->cons->frame) +
            stack_factor);
      cons_list = cons_list->next;
    }
    aux = aux->prev;
  }

  return TRUE;
}
```

- Ante un overflow del Trail + ChoiceStack: En este caso hay que ajustar el valor del punto de elección que se guardó al crear el generador para proteguer a otro generador en la ejecución del backtracking. El codigo que se ejecuta es:

```
CBOOL__PROTO_N(choice_overflow_adjust_tabling_c, int choice_factor)
{
  if (choice_factor == 0)
    return TRUE;
```

```
  struct gen* aux = last_gen_list;

  while (aux) {
    if (aux->node)
      aux->node = (node_t*)((char)(aux->node) + choice_factor);
    aux = aux->prev;
  }

  return TRUE;
}
```

## Aritmética de punteros

Las funciones anteriores reciben como argumento *heap_factor*, *stack_factor* o *choice_factor* respectivamente, Este argumento representa la distancia entre la nueva área de memoria y la anterior. En el caso del Heap se calcula:

```
reloc_factor = (char *)newh - (char *)Heap_Start;
```

Como se puede observar antes de calcular la diferencia se realiza una conversión de tipos a *char* $*$. En el Lenguaje C, los punteros se diferencian según el tipo de dato al que apuntan. De este modo cuando sumamos un número a un puntero lo que hacemos es obtener la dirección de memoria despues de haber avanzado tantos datos de dicho tipo como indica el número que hemos sumado.

*Heap_Start* es un puntero de tipo *tagged_t* $*$ pues apunta a un dato de tipo *tagged_t* que tiene un tamaño de 32 bits y ocupa 4 posiciones de memoria. Si sumamos 5 al puntero *Heap_Start* que apunta, por ejemplo, a la dirección de memoria 70, el resultado es $70 + 5 * 4 = 90$. Sin embargo al convertir *Heap_Start* a un puntero de tipo *char* $*$, el sistema interpreta que el dato al que apunta es de tipo *char* que tienen un tamaño de 8 bits (1 Byte) y ocupa una posición de memoria, el resultado es $70 + 5 * 1 = 75$.

Al mover los datos de una zona de memoria a otra, las variables que apuntan a la zona que hemos movido hay que sumarles el *factor* diferencia. Para realizar dicha suma sin embargo necesitamos que los punteros sean también de tipo *char* $*$. Posteriormnete se realiza otra conversión para asignar a la variable un puntero de su mismo tipo.

```
aux->global_top = (struct tagged_t*)((char *)aux->global_top + heap_factor);
```

## 7.3. Manejo de las áreas de memoria de Tabling

En el módulo de Tabling tenemos dos areas de memoria principales:

- Global Table: Representa la tabla donde guardamos la información principal de Tabling:
  - Los generadores (*struct gen*).
  - Las respuestas asociadas a cada generador (*struct l_ans*).
  - Los nodos del Trie ($TrNode$ y $TrHash$) que ordena los generadores y las respuestas.
- Tabling Stack: Al igual que en los computadores, es la pila donde se guardan las estructuras temporales durante la ejecución de un generador y que al completarse este son liberadas:
  - Consumidores (*struct cons*).
  - Lista de consumidores de un generador (*struct cons_list*).

Al igual que con los stack de la WAM la implementación de Tabling asigna a estas áreas un bloque de memoria suficientemente grande como para no tener que preocuparse de resolver problemas por falta de memoria:

```
#define GLOBAL_TABLE_SIZE       (100000*kCells)
#define TABLING_STK_SIZE        (10000*kCells)
```

Lo cual implica que tenemos $100000 * kCells * 4 = 409600000\ B \approx 390MiB$ para el Global Table y tenemos $39MiB$ para el Tabling Stack. Esta memoria no se crea en el espacio de memoria gestionado por la WAM y tienen que ser gestionadas por mandatos del SO. Lo cual plantea dos cuestiones:

- Las estructuras que guardamos en las áreas de Tabling no están referenciadas por *tagged_t* que como recordaremos son las estructuras que por su limitación de longitud determinaron el tamaño máximo del espacio de memoria de la WAM. Si utilizasemos parte de esta memoria para guardar las estructuras de Tabling estaríamos reduciendo direccionamiento de la WAM con estructuras que podemos referenciar fuera del rango de los *tagged_t*.
- Hacer uso de los mandatos del SO reduce la independencia de Ciao como máquina abstracta. Nos interesa poder utilizar las funciones de gestión de memoria que tenemos ya implementadas para optimizar las solicitudes de aumento de memoria.

Una posible solución a los dos problemas sería replicar la solución del espacio de memoria de la WAM en el módulo de Tabling. Pero mientras que en la WAM el máximo de memoria solicitada venía determinado por el número de direcciones direccionables por el sistema en este caso no tendríamos restricción y la determinación de un tamaño arbitrariamente grande nos condiciona la solución del problema trasladando el problema a un nivel superior en la gestión de memoria.

Dado que las ventajas de ampliar la complejidad en la gestión de memoria implica reproducir los problemas que queremos paliar, la solución planteada implica que las solicitudes de memoria son gestionadas por el SO.

La inicialización de las áreas de memoria se realizan de la siguiente manera:

```
tagged_t *global_table;
tagged_t *tabling_stack;

tagged_t *global_table_free;
tagged_t *tabling_stack_free;
tagged_t *global_table_end;
tagged_t *tabling_stack_end;

#define INIT_GLOBAL_TABLE(GLOBAL_TABLE_SIZE)                                    \
  {                                                                             \
    global_table = (tagged_t*)malloc(GLOBAL_TABLE_SIZE * sizeof(tagged_t*)); \
    global_table_free = global_table;                                           \
    global_table_end = global_table + GLOBAL_TABLE_SIZE;                        \
  }

#define INIT_TABLING_STACK(TABLING_STK_SIZE)                                    \
  {                                                                             \
    tabling_stack = (tagged_t*)malloc(TABLING_STK_SIZE * sizeof(tagged_t*)); \
    tabling_stack_free = tabling_stack;                                         \
    tabling_stack_end = tabling_stack + TABLING_STK_SIZE;                       \
  }
```

Como se puede observar la definición de las áreas de memoria es equivalente a la de la WAM, tenemos registros del inicio del área de memoria, posición actual y última dirección del área de memoria.

El procedimiento mediante el cuál se reserva el espacio de memoria en el en Global Table donde se van a guardar los datos, ya lo mostramos al explicar el sistema de tratamiento de la falta de memoria, mostramos ahora el código para el Tabling Stack que como se puede observar es practicamente idéntico:

```
#define ALLOC_TABLING_STK(PTR,PTR_TYPE,SIZE)            \
  {                                                     \
    (PTR) = (PTR_TYPE)tabling_stack_free;               \
    tabling_stack_free += (SIZE) / sizeof(tagged_t*);   \
                                                        \
    if (tabling_stack_free >= tabling_stack_end) {      \
      fprintf(stderr, "Tabling stack exhausted\n");     \
      SERIOUS_FAULT(" Trying ALLOC_TABLING_STK");       \
    }                                                   \
  }
```

### 7.3.1. Implementación realojando la memoria

La primera aproximación para dotar al modulo de Tabling de una gestión de memoria dinámica consiste en determinar un tamaño de stacks inicial de modo que cuando el sistema requiere más memoria realizar una solicitud de un bloque de memoria mayor, en el que copiar la información y ajustar los punteros. Se trata de reproducir el mecanismo implementado en la WAM.

Para llevar a cabo esta solución tenemos que:

1. Determinar los tamaños iniciales de los stacks y cuanta memoria se va a ir solicitando al ir agotando la memoria disponible.
2. Determinar como se activa el overflow:
   - Se ejecuta overflow al tratar de alojar un dato y comprobar que no hay sitio. Para ello el sistema tiene que estar en situación de poder afrontar un cambio de los datos.
   - El sistema trata de alojar un dato y no hay sitio. Deshace la ejecución hasta una posición estable del sistema y ejecuta el overflow. Finalmente continua con la ejecución.
   - Se determina una zona de seguridad de modo que en las zonas estables de la ejecución se comprueba si estamos dentro de esta zona y en caso afirmativo se ejecuta el overflow.
3. Determinar qué punteros de las estructuras de datos hay que ajustar.

Para determinar el primer punto necesitamos tener la implementación y diseñar una serie de pruebas y una serie de posibilidades de modo que podamos evaluar las ventajas e inconvenientes de las distintas soluciones y escenarios para seleccionar la mejor.

Durante el desarrollo del trabajo se determinaron las dimensiones de los stack de manera arbitraria con la intención de provocar la activación del overflow con ejemplos sencillos. En una fase posterior se incrementaron los tamaños para poder realizar pruebas más complejas.

En relación al segundo punto se determinó seleccionar un area de seguridad y tratar de ajustar las estructuras de datos de modo que el programa continuase con la ejecución sin errores al finalizar el overflow. Al contrario que en la WAM donde se definia un puntero a una dirección de memoria de modo que al sobrepasar dicha posición se activaba el overflow, en nuestra implementación hemos determinado una distancia de modo que si la distancia desde cima del stack hasta el final es menor que la distancia de seguridad se activa el overflow

Para resolver la tercera cuestión fue necesario diseñar una serie de funciones auxiliares para visualizar las estrucutras de datos de Tabling especialmente el Trie.

La implementación que determina si se activa el overflow es la siguiente:

```
if (global_table_free + global_table_warn_offset > global_table_end)
  global_tabling_overflow(Arg);
```

Para determinar donde ubicar esta comprobación tuvimos en consideración que las dos acciones principales que se realizan durante la ejecución de la tabulación son:

- Al ejecutar un predicado tabulado determinar si es generador o consumidor. Esto se hace con la función *tabled_call_c*.
- Al finalizar un predicado tabulado se comprueba si la respuesta es diferente a las que están almacenadas y se guarda. Esto se hace con la función *new_answer_c*.

En un primer momento se colocaron las llamadas a overflow al inicio de las funciones pero mientras que en *tabled_call_c* el sistema era estable y funcionaba correctamente en *new_answer_c* fue necesario colocarla al final. Esto es debido a que *tabled_call_c* es el primer predicado de una regla que se crea para diferenciar los predicados tabulados de los que no lo son y por el contrario *new_answer_c* es el último predicado de dicha regla.

El proceso por el cual se ajustaban los punteros consiste en recorrer los generadores, sus consumidores y los nodos del Trie (*TrNode* y *TrHash*) y ajustar los punteros del stack que se ha realojado.

La implementación de esta solución para el overflow del Global Stack es la siguiente:

```
CVOID__PROTO(global_tabling_overflow)
{
  int count, reloc_factor;
  tagged_t *newh;
  int factor_count = 4, factor_warn = 2;

  count = factor_count * (global_table_end - global_table);

  newh = checkrealloc_ARRAY(tagged_t,
                            count,
                            count*2,
                            global_table);

  reloc_factor = (char *)newh - (char *)global_table;

  global_tabling_overflow_adjust_tabling_gen(Arg, reloc_factor);
  global_tabling_overflow_adjust_tabling_trie(Arg, reloc_factor);

  /* Final adjustments */
  global_table_free = newh + (global_table_free - global_table);
  global_table = newh;
  global_table_end = newh + count;
```

```
  global_table_warn_offset *= factor_warn;
}
```

Se puede observar que en cada llamada se cuatriplica el area de memoria y se dobla la zona de seguridad. Esto es así porque como hemos comentado en el Trie tenemos unas estructuras dentro de los nodos Hash que duplican su tamaño cuando se han llenado de nodos. Lo que implica que según avanze la ejecución mayor será el espacio que tengamos que tener previsto para esta estructura de datos.

Se ejecuta la solicitud de realojar la memoria, se calcula el *factor* igual que en la WAM y se ejecuta el código de ajuste de punteros de los generadores[5] y el Trie[6]. Finalmente se ajustan los punteros que definen el Global Table.

Al realizar las pruebas para evaluar la perdida de rendimento que implicaba realojar la memoria frente a la solución inicial observamos que dado el elevado número de punteros que teníamos que redireccionar la implementación consumia mucho tiempo aumentando el tiempo de ejecución.

### 7.3.2. Implementación añadiendo bloques de memoria

En la WAM el algoritmo de manipulación de datos en la unificación requiere que estos estén dispuestos de manera consecutiva pues se sirve de esta característica para sumando 1 a la dirección de memoria apuntar al siguiente dato.

En el módulo tabling sin embargo los datos son referenciados por punteros y se recorren mediante estructuras enlazadas (listas, tries o hash). Esta situación hace que reajustar todos los punteros sea muy costoso computacionalmente (como hemos podido comprobar con las pruebas).

A su vez la manipulación de los datos en el módulo de tablig se hace siempre consultando la dirección de memoria del siguiente dato que se quiere evaluar. Esta dirección puede ser el siguiente nodo a un nodo del trie si estamos recorriendo el nodo; o puede ser la lista de consumidores de un generador. Esta implementación, nos proporciona la posibilidad de implementar el siguiente mecanismo de gestión de memoria. Se trata de una implementación muy secilla y con un coste computacional mínimo:

- Cuando desde el módulo de tabling se solicita reserva de memria en un stack, se comprueba si hay espacio suficiente.
  - Si hay espacio se devuelve la posición de memoria donde alojar el dato y

[5]El código de la funcioón global_tabling_overflow_adjust_tabling_gen() está en los Anexos
[6]El código de la función global_tabling_overflow_adjust_tabling_trie() está en los Anexos

se continua la ejecución.

- En caso contrario se solicita un nuevo bloque de memoria y se devuelve la posición de memoria del nuevo bloque de memoria como posición para alojar el dato. Dado que no hay necesidad de tener los datos consecutivos en la memoria.

Al tratarse de punteros a direcciones de memoria no tiene importancia que estas no sean del mismo bloque de memoria. Desde el punto de vista de los datos estos están perfectamente referenciados y se puede acceder a ellos y recorrer sin ninguna diferencia.

Evidentemente habría que evaluar que supone el hecho de no tener proximidad referencial. Aunque no lo hemos comentado anteriormente la proximidad referencial es una característica de los datos y las instrucciones de los programas en la que se basan las políticas de gestión de la memoria virtual para reducir el coste de acceder al dispositivo secundario de memoria.

Con esta solución, ya no necesitamos tener en cuenta ninguna de las situaciones del punto 2 con el que iniciabamos el análisis de la gestión del overflow en el módulo de tabling. Cuando se solicita alojar un dato en el Global Table o en el Stack Tabling el sistema recibirá una dirección de memoria en la que dicho dato está almacenado.

El procedimiento realiza los siguientes pasos:

- Guarda en PTR la dirección de memoria libre del stack.
- Se incrementa el valor de la variable que determina cual es la siguiente posición libre del stack considerando reservado el tamaño solicitado.
- Si la variable es mayor que el límite del stack quiere decir que no hay sitio para el tamaño solicitado.
- Se solicita un nuevo bloque de memoria.
- Si la posición de memoria del nuevo bloque es 0x00 quiere decir que no hay memoria disponible en el computador y iniciamos la reinicialización dando el mensaje de error correspondiente.
- En caso de tener una posición de memoria correcta quiere decir que disponemos de un nuevo bloque de memoria, reasignamos a PTR la dirección de memoria libre del nuevo stack.
- Se incrementa el valor de la variable que determina la siguiente posición libre del nuevo bloque de memoria.

El codigo es:

```
#define ALLOC_GLOBAL_TABLE(PTR,PTR_TYPE,SIZE)                    \
  {                                                              \
    (PTR) = (PTR_TYPE)global_table_free;                         \
    global_table_free += (SIZE) / sizeof(tagged_t*);             \
                                                                 \
    if (global_table_free >= global_table_end) {                 \
      INIT_GLOBAL_TABLE_MALLOC(GLOBAL_SIZE);                     \
      if (global_table == 0) {                                   \
        SERIOUS_FAULT("␣Trying␣ALLOC_GLOBAL_TABLE");             \
      }                                                          \
      (PTR) = (PTR_TYPE)global_table_free;                       \
      global_table_free += (SIZE) / sizeof(tagged_t*);           \
    }                                                            \
  }

#define ALLOC_TABLING_STK(PTR,PTR_TYPE,SIZE)                     \
  {                                                              \
    (PTR) = (PTR_TYPE)tabling_stack_free;                        \
    tabling_stack_free += (SIZE) / sizeof(tagged_t*);            \
                                                                 \
    if (tabling_stack_free >= tabling_stack_end) {               \
        INIT_TABLING_STACK(STACK_SIZE);                          \
      if (tabling_stack == 0) {                                  \
        SERIOUS_FAULT("␣Trying␣ALLOC_TABLING_STK");              \
      }                                                          \
      (PTR) = (PTR_TYPE)tabling_stack_free;                      \
      tabling_stack_free += (SIZE) / sizeof(tagged_t*);          \
    }                                                            \
  }
```

Donde $GLOBAL\_SIZE$ y $STACK\_SIZE$ son los tamaños de los nuevos bloques que en la macro de inicialización podemos ir incrementando como hacíamos en la solución A o podemos optar por solicitar siempre bloques del mismo tamaño.

Como se puede observar dado que en la mayoria de las ocasiones se podrá asignar la posición de memoria solicitada, se realiza la asignación del puntero y se comprueba si nos hemos salido del bloque de memoria asignado (esta operación no supone un incremento en el rendimiento respecto a la implementación inicial en la que tambien se realizaba esta comprobación):

- Si el dato cabe dentro del bloque de memoria se da por buena la asignación y la ejecución continua.
- En caso contrario realizamos la solicitud de un nuevo bloque de memoria:
  - En caso de que no tengamos memoria disponible en el sistema el puntero de la nueva zona de memoria será el 0x00 por lo tanto en esa situación ejecutamos el mecanismo de reinicialización.
  - Si por el contrario hemos obtenido un nuevo bloque de memoria, repetimos la operación inicial para alojar el dato en el nuevo bloque de memoria y esta es la dirección de memoria del dato almacenado en el stack

# Capítulo 8

# PRUEBAS

En el presente capítulo se detallan las pruebas realizadas durante la implemneta-ción de las funciones descritas en los capítulos anteriores, así como los resultados obtenidos.

Las pruebas están agrupadas en tres apartados:

- Prueba de captura de errores en Tabling: con estas pruebas hemos evaluado que el modulo de Tabling es capaz de detectar la falta de memoria y ejecutar el procedimiento de reinicialización junto con la WAM.
- Prueba del manejo de la memoria de Ciao: con estas pruebas hemos evaluado que cuando la WAM ejecuta el procedimiento ante un desbordamiento de memoria, el modulo de Tabling es capaz de continuar con la ejecución.
- Prueba del manejo de la memoria de Tabling: con estas pruebas hemos evaluado que Tabling ejecuta correctamente la gestión dinámica de sus áreas de memoria. Igualmente hemos realizado un conjunto de pruebas en las que hemos medido el tiempo empleado con distintas soluciones en la gestión de memoria.

Durante la evaluación de las pruebas, ha sido necesario analizar el comportamiento del programa y para ello hemos utilizado el depurador lldb (equivalente al gdb) que incorpora el sistema operativo OX de Apple.

Para poder monitorizar la ejecución del programa con el depurador es necesario compilar el codigo de Ciao con el flag $-g$ sin embargo dado que la compilación se realiza mediante un script, hay que configurar los parámetros de dicho script mediante:

$$./ciaosetupconfigure - -interactive$$

A su vez dado que parte del codigo de Tabling se compila a partir de un modulo de prolog (tabling_rt.pl) hay que especificar en dicho archivo que tambien queremos el

activar el flag de compilación con depuración:

$$:-extra\_compiler\_opts(['-g']).$$

A la vez que activamos el flag de compilación con depuración, se declara la variable *DEBUG* de modo que se compila código auxiliar de depuración que nos permite realizar trazas de ejecución del programa. Este código auxiliar no se compila en la versión comercial dado que los procedimientos de traza reducen el rendimiento del programa.

Las pruebas se han realizado en un MacBook Pro de Apple con:

El sistema operativo OS X version 10.9.2 (Maverick).
Con un procesador 2.66 Ghz (Intel Core 2 Duo).
Una memoria de 4 GB (1067 Mhz DDR3).
Pantalla de 15çon tarjeta gráfica NVIDIA GeForce 9400M de 256 MB.

## 8.1. Pruebas de captura de errores en Tabling

Para las pruebas por falta de memoria se planteaban dos posibilidades:

- Diseñar una serie de programas que agotasen respectivamente los Stacks de la WAM y activasen el mecanismo de reinicialización de la WAM.
- Diseñar un procedimiento que mediante un flag impidiese ejecutar el overflow de los stacks y activase el mecanismo de reinicialización de la WAM.

Finalmente realizamos las pruebas utilizando la segunda posibilidad. Para ello inicialmente implementamos en Ciao unos flags que podiamos manipular con los siguientes predicados:

```
set_prolog_flag/2.
current_prolog_flag/2
```

Sin embargo esta solución implicaba aumentar la dependencia en el código de la Wam respecto a Tabling lo cual iba en contra del concepto de independencia de los módulos.

Para resolver esto implementamos unos predicados en el modulo de Tabling de modo que se definían e inicializaban en el código de Tabling como se puede consultar en los Anexos. Los predicados para manipular los flags de tabling son:

```
set_tabling_flag/2.
current_tabling_flag/2
```

Los flags definidos he implementados para la realización de pruebas y depuración son:

```
   if (!tabling_adjust)     tabling_adjust = atom_on;
   if (!tabling_overflow)   tabling_overflow = atom_off;
   if (!tabling_increase)   tabling_increase = atom_off;
#if defined(DEBUG)
   if (!tabling_bypass)     tabling_bypass = atom_off;
   if (!tabling_debug)      tabling_debug = atom_off;
   if (!tabling_trace)      tabling_trace = atom_off;
   if (!tabling_print)      tabling_print = atom_off;
#endif
```

El codigo que provoca la reinicialización de la WAM sin necesidad de llenar las áreas de memoria es:

```
#if defined(TABLING)
#if defined(DEBUG)
   if (tabling_bypass == atom_on) {
     if (tabling_debug == atom_on)
       printf("Bypass heap overflow\n");
     SERIOUS_FAULT(" Trying heap overflow - bypass ");
   }
#endif
#endif
```

Evidentemente provocar que la WAM reinicie es una operación solo necesaria en modo depuración por lo que solo se compila al seleccionar depuración.

## 8.2. Pruebas del manejo de la memoria de Ciao

En este caso también se hace uso de un flag de tabling (*adjust*) de modo que en el archivo que activa el modulo de Tabling *tabling_rt.pl* podemos especificar si queremos activar (*on*) la gestión dinámica de las áreas de memoria de la WAM reajustando los punteros de Tabling o si por el contrario se quiere ejecutar Tabling con la solución implementada anteriormente.

En esta ocasión se hacía necesario diseñar un programa que utilizase tabulación e hiciese uso intensivo de las áreas de memoria de la WAM.

El siguiente programa es un bucle tabulado que genera como salida $R$ un término de la aritmética de Peanno equivalente al valor de entrada $N$.

```
:- module(_, _).
:- use_package(library(tabling)).
:- table p/2.
p(N, R) :-
        N > 0,
        N1 is N - 1,
        p(N1, O),
        R = s(O).
p(0, z).
```

```
p(0, a).
```

que funciona correctamente con las siguientes llamadas:

```
Ciao 1.15-2092-g2b79834 (2014-05-20 13:12:12 +0200) [DARWINi86]  [debug]
?- use_module('/Users/joaquin/Documents/Imdea/CiaoRepositorio/ciao-devel4/
    ciao/contrib/tabling/stack_expansion_examples/ex01.pl').
Tabling_adjust      active

yes
?- p(1000,_).
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow

yes
?- p(2000,_).

yes
?- p(4000,_).
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow

yes
?-
```

Con un ejemplo de generadores anidados del articulo sobre CHAT [5].

```
:- module(_, _).
:- use_package(library(tabling)).
%Paper Chat - Kostis
:-table t/1.
query(Choices,Consumers) :-
        make_choices(Choices,_),
        make_consumers(Consumers,[]).
make_choices(N,trail) :-
        N>0, M is N-1, make_choices(M,_).
make_choices(0,_).
make_consumers(N,Acc) :-
        N>0, M is N-1,
        t(_),
        make_consumers(M,[a|Acc]).
make_consumers(0,_).
t(1).
```

La ejecución del programa provoca el desbordamiento de las áreas de memoria de la Wam y no falla durante la ejecución:

```
Ciao 1.15-2092-g2b79834 (2014-05-20 13:12:12 +0200) [DARWINi86]  [debug]
?- use_module('/Users/joaquin/Documents/Imdea/CiaoRepositorio/ciao-devel4/
    ciao/contrib/tabling/stack_expansion_examples/ex01.pl').
Tabling_adjust      active

yes
?- query(100000,100000).
Llamando choice_overflow_adjust_tabling desde choice_overflow
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow
```

```
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando heap_overflow_adjust_tabling desde heap_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow
Llamando heap_overflow_adjust_tabling desde heap_overflow
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow
Llamando stack_overflow_adjust_tabling desde stack_overflow
Llamando heap_overflow_adjust_tabling desde heap_overflow
Llamando choice_overflow_adjust_tabling desde choice_overflow

yes
?-
```

## 8.3. Pruebas del manejo de la memoria de Tabling

Para realizar las pruebas de la primera solución que ampliaba memoria en Tabling utilizamos el flag de tabling (*overflow*) que creaba unas áreas de memoria en Tabling de reducido tamaño con el fin de llegar a llenarlas con un sencillo programa. Las diferentes pruebas permitieron ir completando los ajustes que se tenían que realizar en las estructuras de Tabling.

Para comprender debidamente como se comportaba la implementación de Tabling se hizo necesario implementar diversas funciones que imprimian los datos de las estructuras. Estas mismas funciones las conectamos con predicados en Prolog para poder mostrar el estado de los datos de Tabling durante la ejecución de los programas.

Finalmente cuando el proceso funcionaba correctamente organizamos los casos de pruebas de modo que mostrasen el tiempo de ejecución. El predicado *spend_time_.../2* calculaba el tiempo de ejecución al procesar todas las respuestas del predicado correspondiente:

```
:- use_module(library(prolog_sys)).
spend_time_p(N, Time) :-
        statistics(runtime, _),
        ( p(N,P), display(P), nl, fail; true ),
          statistics(runtime, [_, Time]).
```

Para poder evaluar la implicación de los cambios realizados hicimos uso de los flags de tabling implementados activando y desactivando las funcionalidades implementadas:

```
:- initialization(set_tabling_flag(adjust, on)).
:- initialization(set_tabling_flag(overflow, off)).
```

El codigo utilizado para realizar las pruebas está en los Anexos.

Se ejecutan cuatro llamadas en tres escenarios diferentes:

- Las llamadas son:
  1 - $pTabled(300, P)$
  2 - $fibonacciTabled(30, P)$ -
  3 - $pTabled\_Fibonacci(25, P)$
  4 - $pTabled\_Fibonacci\_20(100, P)$

- Los escenarios son:
  A - Prolog
  B - Prolog + Tabling
  C - Prolog + Tabling + Realojar memoria
  D - Prolog + Tabling + Bloques de memoria

El resultado queda resumido en las siguientes tablas:

| | **A** | | | | **D** | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | WAM | | | Time | WAM | | | Tabling | Time |
| | H | S | C | | H | S | C | | |
| **pTabled(300,P)** | 0 | 0 | 0 | **0.662** | 0 | 0 | 0 | 9 | **14.5553** |
| **fibonacciTabled(30,P)** | 0 | 11 | 11 | **697.928** | 0 | 0 | 0 | 1 | **0.478** |
| **pTabled_Fibonacci(25,P)** | 0 | 9 | 9 | **143.425** | 0 | 9 | 9 | 0 | **170.788** |
| **pTabled_Fibonacci_20(100,P)** | 0 | 11 | 11 | **602.742** | 0 | 11 | 11 | 3 | **606.577** |

| | **B** | **C** | | **D** | |
|---|---|---|---|---|---|
| | Time | Tabling | Time | Tabling | Time |
| **pTabled(300,P)** | **13.885** | 9 | **15.567** | 56 | **14.553** |
| **fibonacciTabled(30,P)** | **0.413** | 1 | **0.516** | 1 | **0.478** |
| **pTabled_Fibonacci(25,P)** | **168.107** | 0 | **170.893** | 0 | **170.788** |
| **pTabled_Fibonacci_20(100,P)** | **ERROR:** | 3 | **607.614** | 0 | **606.577** |

La primera tabla no es concluyente para el trabajo realizado pero se muestra como información descriptiva del funcionamiento de los dos métodos de resolución descritos en el trabajo (SLD y OLDT):

- pTabled/2 es un bucle que no aprovecha las ventajas de la tabulación porque cada llamada del predicado tabulado es nueva y esto implica tener que guardarla aumentando el tiempo de ejecución del programa.

- fibonacciTabled/2 sin embargo ilustra las ventajas de utilizar tabulación cuando se repiten llamadas.
- pTabled_Fibonacci/2 repite las desventajas del primer programa aunque solo se hacen 25 llamadas frente a las 300 de pTabled/2.
- pTabled_Fibonacci_20/2 De nuevo un programa que penaliza la tabulación.

En la segunda tabla podemos comparar resultados en los que es determinante el uso de los gestores de memoria:

- pTabled/2: el programa ejecutado sin gestor de memoria (B) es más rápido pero podemos comprobar que realizando hasta 56 peticiones de bloques de memoria en el caso D somos más rápidos que en el caso C.
- fibonacciTabled/2: Se confirma que la solución D es más rápida que la C.
- pTabled_Fibonacci/2: El ajuste de los punteros de la WAM evidentemente penalizan a los casos C y D frente a B. Al no haber realojos en Tabling los tiempos de C y D son similares.
- pTabled_Fibonacci_20/2: La implementación inicial de tabling no puede ejecutar el programa porque se queda sin memoria.

La nueva implementación se adapta a las necesidades de memoria del programa con un coste computacional que hemos tratado de minimizar.

# Capítulo 9

# CONCLUSIONES

El uso de los lenguajes de programación de alto nivel facilita el trabajo de los programadores al ofrecerles un lenguaje más cercano al problema que se quiere resolver. A su vez estos lenguajes de programación permiten al programador abstraerse de problemas como la gestión de memoria.

El enunciado del presente trabajo **manejo de memoria en programación lógica con tabulación** resume la esencia del objetivo del desarrollo de los lenguajes de programación de alto nivel:

- Manejo: manejar, gestionar, no se refiere solo a resolver los casos de funcionamiento correcto, se trata también de manejar las sitiaciones anómalas que provocan que un programa falle.
- de memoria: La gestión de la memoria es trascendental no solo en programación, también en el diseño de la arquitectura de los computadores, en 1995 en [14] se planteaba el problema del "Memory Wall".
- en programación: La gestión explicita de la memoria en lenguajes como C permiten optimizar su uso para incrementar el rendimiento de los computadores pero incrementan la complejidad y con ello la probabilidad de errores.
- lógica: es un paradigma de programación basado en la lógica de $1^{er}$ orden que permite al programador expresar el problema con un lenguaje muy proximo al problema que pretende modelizar.
- con tabulación: es un método de resolución que incrementa las posibilidades descriptivas de Prolog (lenguaje de programación lógico) y permite aumentar la velocidad de ejcución de los programas aumentando los requerimientos de memoria.

La implementación presentada en este trabajo aborda en esencia:

- La necesidad de gestionar fallos durante la ejecución y reinciar el sistema.
- Compatibilizar el uso de la tabulación con la gestión dinámica de las áreas de memoria de Ciao.
- Dotar al módulo de tabling de un sistema de gestión de memoria dinámico.

Para el desarrollo del presente trabajo ha sido necesario:

- Comprender y saber utilizar el lenguaje de programación Prolog.
- Tener conocimientos de C, lenguaje en el que está implementado Ciao Prolog.
- Utilizar el depurador para analizar el comportamiento del programa.
- Comprender el funcionamiento del mecanismo de resolución de Prolog, SLD y el de la tabulación.
- Conocer las estructuras de datos de la WAM y del módulo de tabling.
- Determinar como acceder a funciones del módulo desde la WAM.
- Y otros muchos detalles que he tratado de ordenar en el presente documento

El resultado final de la implementación ha tratado de garantizar que el funcionamiento de Ciao Prolog no se ve afectado:

- La implementación propuesta para realizar el reincio del módulo de tabling no provoca overhead pues se ejecuta junto con la reinicialización de la WAM.
- El ajuste de los punteros de tabling cuando se produce overflow en la WAM solo se ejecuta cuando ha sido necesario realizar el realojo de memoria. Si tabling está cargado pero no se ejecuta su overhead es mínimo en comparación con el coste del overflow pues se consume el tiempo de ejecutar una función como no tiene datos que recorrer finaliza su ejecución inmediatamente.
- La solución de solicitar más bloques de memoria cuando hay desbordamiento en el módulo de tabling, hace las mismas comprobaciones que se realizaba antes para detectar un desbordamiento de modo que el overhead es nulo.

# Capítulo 10

# TRABAJO FUTURO

En el presente capítulo exponemos las líneas de trabajo detectadas para continuar con la mejora en la implementación del módulo tabling e indicamos problemas de la implementación desarrollada.

## Reinicialización

La implementación de *reinit_tabling* si bien cumple los objetivos previstos ha generado dos nuevas tareas que deberán acometerse en el futuro para continuar con la mejora del comportamiento de Ciao:

1. Acceso a funciones C de los módulos de manera más eficaz desde el punto de vista de reducir el tiempo de ejecución así como evitar la necesidad de definir un predicado Prolog para funcionar con funciones C. No parece buena práctica acceder desde C a un procedimiento Prolog para ejecutar otro procedimiento C. Sería deseable que esta conexión se resolviese en la capa de implementación de C.
2. Detectar el porque del valor inesperado de $goal\_desc->action$ y encontrar la solución que no haga necesario el parche que hemos creado.

## Desbordamiento de memoria de la WAM

La implementación de $..._overflow_adjust_tabling$ si bien cumple los objetivos previstos ha generado dos nuevas tareas que deberán acometerse en el futuro:

1. Al analizar las estrucutras de datos se han detectado atributos que podrían ser redundantes. Una tarea que aumentaria el rendimiento desde el punto de vista

de consumo de memoria es evidentemente optimizar el tamaño de las estrucutras de datos de manera que solo contengan la información imprescindible.

2. Hacer compatible Tabling con la recolección de basura (Garbage Collector) de la WAM, que en estos momentos se desactiva al cargar el módulo de Tabling:

```
:- initialization(nogc).
```

   La ejecución del GC se ejecuta ante la falta de memoria en los stacks y libera de la memoria aquella información que ya no es requerida. Es entonces, si la memoria liberada no es suficiente cuando se solicita la ampliación del área de memoria. Sin embargo la ejecución del GC implica que las estructuras cambian su posición en memoria y se requiere algún mecanismo para modificar aquellos punteros de Tabling que se ven afectados.

La solución de estos problemas puede implicar que la idea de que Tabling sea un modulo independiente al *engine* de la WAM tenga que ser reconsiderado.

A su vez durante las pruebas una invocación alta del predicado $p(6000, P)$. provocaba el siguiente error:

```
* thread #1: tid = 0x26ecc, 0x0017a1ae tabling_rt_DARWINi86.dylib`
    nd_resume_cons_c(w=0x6ffddc50) + 222 at chat_tabling.c:985, queue = 'com.
    apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1, address=0
    xffdf9398)
    frame #0: 0x0017a1ae tabling_rt_DARWINi86.dylib`nd_resume_cons_c(w=0
        x6ffddc50) + 222 at chat_tabling.c:985
   982    if (icons_l != NULL)
   983      {
   984        //Link shared trail of children consumers
-> 985        if (LastNodeTR != icons_l->cons->node_tr) // XS: segmentation
    violation pFact(100,_) after reinit
   986          {
   987            LastNodeTR->next = icons_l->cons->node_tr;
   988            //it was linked to previous LastNodeTR
```

# Desbordamiento en el módulo de tabling

La implementación del sistema de overflow en el módulo de tabling mediante la solicitud de nuevos bloques de memoria ha generado nuevas posibiildades de implementación que hay que definir, implementar y evaluar para determinar cual es la solución más adecuada para:

- La implementación del desalojo de la memoria guardad en el Stack Tabling al finalizar la ejecución de un generador. Dado que los bloques no están conectados, el procedimiento actual lo que hace es indicar como siguiente posición libre del stack la posición donde empezaron a guardarse datos del generador que se ha completado.:

- La solución actual sin ninguna modificación, provoca que los bloques creados después del bloque donde está esta posición de memoria quedan sin referenciar. Sería memoria perdida.
- Evidentemente es necesario llevar un registro de los bloques solicitados. La mejor solución es un stack. Pero en este caso tenemos dos alternativas:
  * Conservar las direcciones de los bloques creados y cuando volvamos a necesitar memoria utilizar dichos bloques. Sería equivalente a la solución implementada en la WAM.
  * Devolver al sistema los bloques de memoria. Esta solución tiene un pequeño overhead pero sin duda menor que el que tendría en la implementación de la WAM donde restaurar el tamaño de los stacks cuando ya no es necesaria la memoria solicitada implica reajustar los punteros.
- La implementación del borrado del Global Table. En este caso es un predicado que invoca el programador directamente cuando quiere limpiar los datos de la tabulación. La problematica y las opciones son las mismas que para el caso anterior.

# Bibliografía


[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compiler: Principles, Techniques, and Tools (the 'Red Dragon Book')*. Pearson Education, Inc, first edition, 1986.

[2] Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconsruction*. MIT Press, 1991.

[3] Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for prolog based on wam. *Commun. ACM 31(6): 719-741*, 1988.

[4] Pablo Chico de Guzmán Huerta. *Estrategias Avanzadas de Tabulación y Paralelismo en Programas Lógicos*. PhD thesis, Universidad Politécnica de Madrid, 2012.

[5] Bart Demoen and Konstantinos F. Sagonas. Chat: The copy-hybrid approach to tabling. *In Practical Applications of Declarative Languages: 106–121*, 1999.

[6] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming (TPLP), 12: 219-252*, 2012.

[7] John W. Lloyd. Practical advantanges of declarative programming. *GULP-PRODE(1): 18-30*, 1994.

[8] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *Logic Programing, 1(38): 31-54*, 1999.

[9] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM 12(1): 23–41*, 1965.

[10] Sterling and Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.

[11] H. Tamaki and T. Sato. Old resolution with tabulation. *Third International Conference on Logic Programming: 84-98*, 1986.

[12] John von Neumann. *First draft of a Report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[13] David H. D. Warren. An abstract prolog instruction set. *Technical Note 309, SRI International*, 1983.

[14] W. Wulf and S. McKee. Hitting the wall: Implications of the obvious. *ACM SIGArch Computer Architecture News, 23(1):20–24*, 1995.

# ANEXOS

## 1. statistics_resumen()

```
CVOID__PROTO(statistics_resume) {
  int total, use;
  float  percentual;
  frame_t *newa;

  total = 4 * (int)HeapDifference(Heap_Start, Heap_End);
  use = 4 * (int)HeapDifference(Heap_Start, w->global_top);
  percentual = 100.0 * use / total;
  printf("Heap   (%10d - %10d) in use %8d of %10d (%2.2f %%)\n",
         (unsigned int)Heap_Start,
         (unsigned int)Heap_End,
         use,
         total,
         percentual);

  ComputeA(newa,w->node);
  total = 4 * (int)StackDifference(Stack_Start, Stack_End);
  use = 4 * (int)StackDifference(Stack_Start,newa);
  percentual = 100.0 * use / total;
  printf("Stack  (%10d - %10d) in use %8d of %10d (%2.2f %%)\n",
         (unsigned int)Stack_Start,
         (unsigned int)Stack_End,
         use,
         total,
         percentual);

  total = 4 * (int)TrailDifference(Trail_Start, Trail_End);
  use = 4 * (int)TrailDifference(Trail_Start, w->trail_top);
  percentual = 100.0 * use / total;
  printf("Trail  (%10d - %10d) in use %8d of %10d (%2.2f %%)\n",
         (unsigned int)Trail_Start,
         (unsigned int)Trail_End,
         use,
         total,
         percentual);

  total = 4 * (int)ChoiceDifference(Choice_Start, Choice_End);
  use = 4 * (int)ChoiceDifference(Choice_Start, w->node);
  percentual = 100.0 * use / total;
  printf("Choice (%10d - %10d) in use %8d of %10d (%2.2f %%)\n",
         (unsigned int)Choice_Start,
         (unsigned int)Choice_End,
         use,
         total,
         percentual);
}
```

# 2. global_tabling_overflow_adjust_tabling_gen()

```
CVOID__PROTO_N(global_tabling_overflow_adjust_tabling_gen, int reloc_factor)
{
  struct gen* aux;
  struct l_ans* aux_ans;

  last_gen_list = (struct gen*)((char *)last_gen_list + reloc_factor);

  aux = last_gen_list;

  while(aux) {
    aux->leader = (struct gen*)((char *)(aux->leader) + reloc_factor);
    if (aux->ptcp) aux->ptcp = (struct gen*)((char *)(aux->ptcp) + reloc_factor
        );

    if (aux->first_cons) // this is a pointer to Stack Tabling
      global_tabling_overflow_adjust_tabling_cons(Arg, aux->first_cons,
          reloc_factor);

    aux->trie_ans = (trie_node_t*)((char *)(aux->trie_ans) + reloc_factor);
    aux_ans = aux->first_ans;
    while(aux_ans) {
      if (aux_ans->node) aux_ans->node = (TrNode)((char *)aux_ans->node +
          reloc_factor);
      if (aux_ans->next) aux_ans->next = (struct l_ans *)((char *)aux_ans->next
           + reloc_factor);
      aux_ans = aux_ans->next;
    }
    if (aux->first_ans)
      aux->first_ans = (struct l_ans*)((char *)(aux->first_ans) + reloc_factor)
          ;
    if (aux->last_ans)
      aux->last_ans = (struct l_ans*)((char *)(aux->last_ans) + reloc_factor);
    if (aux->prev) aux->prev = (struct gen*)((char *)(aux->prev) + reloc_factor
        );
    if (aux->post) aux->post = (struct gen*)((char *)(aux->post) + reloc_factor
        );
    aux = aux->prev;
  }

  for (int i=iptcp_stk-1; i>0; i--) {
    if (ptcp_stk[i])
      ptcp_stk[i] = (struct gen *)((char *)ptcp_stk[i] + reloc_factor);
  }
}

CVOID__PROTO_N(global_tabling_overflow_adjust_tabling_cons, struct cons_list*
    cons_list, int reloc_factor)
{
  struct cons_list* aux = cons_list;

  while (aux) {
    if (aux->cons->last_ans) {
      aux->cons->last_ans = (struct l_ans *)((char *)aux->cons->last_ans +
          reloc_factor);
    }
    aux->cons->ptcp = (struct gen *)((char *)aux->cons->ptcp + reloc_factor);
    aux->cons->gen = (struct gen *)((char *)aux->cons->gen + reloc_factor);
    aux = aux->next;
  }
}
```

# 3. global_tabling_overflow_adjust_tabling_trie()

```
CVOID__PROTO_N(global_tabling_overflow_adjust_tabling_trie, int reloc_factor)
{

  trie_node_top = (TrNode)((char *)trie_node_top + reloc_factor);

  global_tabling_overflow_adjust_tabling_trie_trNode(Arg, trie_node_top,
      reloc_factor);

}

CVOID__PROTO_N(global_tabling_overflow_adjust_tabling_trie_trNode, TrNode
    trie_node, int reloc_factor)
{

  if (! IS_TRIE_HASH(trie_node))
    {
      if (trie_node->next) {
        if (trie_node->next != (TrNode)1 && trie_node->next != (TrNode)2) {
          trie_node->next = (TrNode)((char *)trie_node->next + reloc_factor);
          global_tabling_overflow_adjust_tabling_trie_trNode(Arg, trie_node->
              next, reloc_factor);
        } else {
          return;
        }
      }

      if (trie_node->child) {
        if (trie_node->child != (TrNode)1 && trie_node->child != (TrNode)2) {
          trie_node->child = (TrNode)((char *)trie_node->child + reloc_factor);
          global_tabling_overflow_adjust_tabling_trie_trNode(Arg, trie_node->
              child, reloc_factor);
        } else {
          return;
        }
      }
    }
  else     // trie_node IS_TRIE_HASH
    {
      TrHash hash = (TrHash)trie_node;

      hash->buckets = (TrNode *)((char *)hash->buckets + reloc_factor);
      for (int i=0; i < hash->number_of_buckets; i++) {
        if (hash->buckets[i]) {
          hash->buckets[i] = (TrNode)((char *)hash->buckets[i] + reloc_factor);
          global_tabling_overflow_adjust_tabling_trie_trNode(Arg, hash->buckets
              [i], reloc_factor);
        }
      }
    }
}
```

# 4. set_tabling_flag/2 y current_tabling_flag/2

```
CBOOL__PROTO(set_tabling_flag_c) {
#if defined(TABLING)
#if defined(DEBUG)
  if (tabling_debug == atom_on)
    printf("Set_tabling_flag\n"); //XS
#endif
```

```
  char * flag, * value;
  tagged_t newvalue;

  /* check value is on or off */

  DEREF(ARG2,ARG2);
  value = (char *)TagToAtom(ARG2)->name;

  if (strcmp(value, "off")==0) newvalue = atom_off;
  else if (strcmp(value, "on")==0) newvalue = atom_on;
  else  return FALSE;

  /* check flag is correct */

  DEREF(ARG1,ARG1);
  flag = (char *)TagToAtom(ARG1)->name;

  if (strcmp(flag, "adjust")==0) tabling_adjust = newvalue;
  else if (strcmp(flag, "overflow")==0) tabling_overflow = newvalue;
  else if (strcmp(flag, "increase")==0) tabling_increase = newvalue;
#if defined(DEBUG)
  else if (strcmp(flag, "debug")==0) tabling_debug = newvalue;
  else if (strcmp(flag, "debug_tries")==0) tabling_debug_tries = newvalue;
  else if (strcmp(flag, "trace")==0) tabling_trace = newvalue;
  else if (strcmp(flag, "print")==0) tabling_print = newvalue;
  else if (strcmp(flag, "bypass")==0) tabling_bypass = newvalue;
#endif
  else return FALSE;

  return TRUE;

#else
  printf("\nTABLING Flag must be activated\n");
  return FALSE;
#endif
}

// TODO XS: current_tabling_flag can NOT be used to request flags as
    current_prolog_flag(X,Y).
CBOOL__PROTO(current_tabling_flag_c) {
#if defined(TABLING)
#if defined(DEBUG)
  if (tabling_debug == atom_on)
    printf("Set_tabling_flag\n"); //XS
#endif

  char * flag, * value;
  tagged_t newvalue;

  /* check flag is correct */

  DEREF(ARG1,ARG1);
  if (IsAtom(ARG1))
    flag = (char *)TagToAtom(ARG1)->name;
  else return FALSE;


  if (strcmp(flag, "adjust")==0) return Unify(ARG2,tabling_adjust);
  else if (strcmp(flag, "overflow")==0) return Unify(ARG2,tabling_overflow);
  else if (strcmp(flag, "increase")==0) return Unify(ARG2,tabling_increase);
#if defined(DEBUG)
  else if (strcmp(flag, "debug")==0) return Unify(ARG2,tabling_debug);
  else if (strcmp(flag, "debug_tries")==0) return Unify(ARG2,
      tabling_debug_tries);
  else if (strcmp(flag, "trace")==0) return Unify(ARG2,tabling_trace);
  else if (strcmp(flag, "print")==0) return Unify(ARG2,tabling_print);
```

```
  else if (strcmp(flag, "bypass")==0) return Unify(ARG2,tabling_bypass);
#endif
  return FALSE;

#else
  printf("\nTABLING Flag must be activated\n");
  return FALSE;
#endif
}
```

# 5. pruebas.pl

```
:-module(_,_).

:- use_package(library(tabling)).

:- table p/2.
p(N, R) :-
        N > 0,
        N1 is N - 1,
        p(N1, O),
        R = s(O).
p(0, z).
p(0, a).

:- fibonacci/2.
fibonacci(0,0).
fibonacci(1,1).
fibonacci(X,Y) :-
        X > 1,
        X1 is X - 1,
        X2 is X - 2,
        fibonacci(X1, Y1),
        fibonacci(X2, Y2),
        Y is Y1 + Y2.

:- pFibonacci/2.
pFibonacci(N, R) :-
        N > 0,
        fibonacci(N,_),
        N1 is N - 1,
        pFibonacci(N1, O),
        R = s(O).
pFibonacci(0, z).
pFibonacci(0, a).

:- use_module(library(prolog_sys)).
spend_time_p(N, Time) :-
        statistics(runtime, _),
        ( p(N,P), display(P), nl, fail; true ),
          statistics(runtime, [_, Time]).

spend_time_fibonacci(N, Time) :-
        statistics(runtime, _),
        ( fibonacci(N,P), display(P), nl, fail; true ),
          statistics(runtime, [_, Time]).

spend_time_pFibonacci(N, Time) :-
        statistics(runtime, _),
        ( pFibonacci(N,P), display(P), nl, fail; true ),
          statistics(runtime, [_, Time]).
```

Este documento esta firmado por

| | | |
|---|---|---|
|  | **Firmante** | CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES |
| | **Fecha/Hora** | Wed Jun 11 17:35:43 CEST 2014 |
| | **Emisor del Certificado** | EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES |
| | **Numero de Serie** | 630 |
| | **Metodo** | urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature) |